Introduction

Yaga2 is an ML-based anomaly detection system for monitoring service health in production environments. This guide explains how the system detects anomalies, evaluates their operational significance, and manages incident lifecycles.

What Problem Does Yaga2 Solve?

Modern microservice architectures generate massive amounts of metrics data. Manually setting static thresholds for every service and metric combination is:

- **Impractical**: Hundreds of services × multiple metrics × different time periods = thousands of thresholds
- **Brittle**: Traffic patterns change seasonally, after deployments, and as services evolve
- **Noisy**: Static thresholds either miss real issues or generate alert fatigue

Yaga2 addresses these challenges by:

1. **Learning normal behavior** from historical data using machine learning
2. **Detecting deviations** that represent genuine anomalies, not just threshold breaches
3. **Evaluating operational impact** to determine if anomalies actually matter
4. **Managing alert lifecycle** to reduce noise and prevent duplicate alerts

Core Concepts

What is Anomaly Detection?

Anomaly detection identifies data points that deviate significantly from expected patterns. Unlike threshold-based alerting (e.g., “alert if latency > 500ms”), anomaly detection learns what “normal” looks like for each service and flags when behavior deviates from that baseline.

Example:

- Service A normally has 200ms latency → 350ms is anomalous
- Service B normally has 800ms latency → 350ms is actually *better* than normal
- A static 500ms threshold would miss the anomaly for Service A and false-alarm for Service B

What is an SLO?

SLO (Service Level Objective) is a target level of service reliability. It defines what “good enough” means for your service.

Common SLOs include:

- **Latency SLO**: “99% of requests complete in under 500ms”
- **Availability SLO**: “Service is available 99.9% of the time”
- **Error Rate SLO**: “Less than 0.5% of requests result in errors”

Why SLOs Matter for Anomaly Detection:

An anomaly might be statistically significant but operationally irrelevant:

- ML detects latency at 280ms (unusual compared to baseline of 150ms)
- But SLO says 500ms is acceptable
- Result: The anomaly is real but doesn’t warrant immediate action

Yaga2 combines ML detection with SLO evaluation to answer two questions:

1. **Is this unusual?** (ML detection)
2. **Does it matter?** (SLO evaluation)

What is Isolation Forest?

Isolation Forest is the machine learning algorithm at the heart of Yaga2's anomaly detection. It's an "unsupervised" algorithm, meaning it doesn't need labeled examples of anomalies—it learns what's normal from your data and identifies deviations.

The key insight: **Anomalies are easier to isolate than normal points.**

Think of it like a game of "20 questions" for data points:

- Normal data points are similar to many others, requiring many questions to identify uniquely
- Anomalies are outliers that can be identified with just a few questions

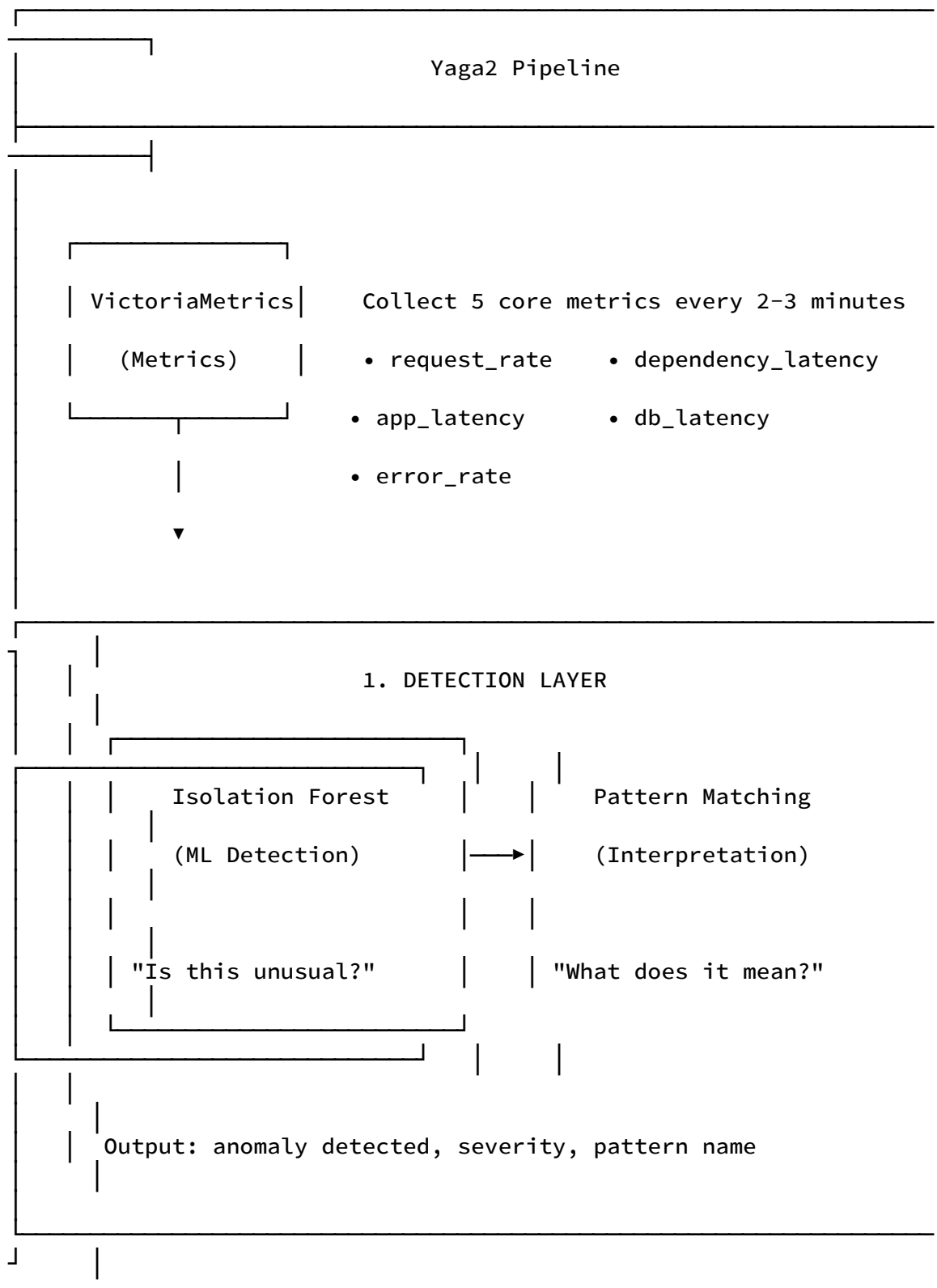
We'll explore Isolation Forest in detail in the Detection Layer section.

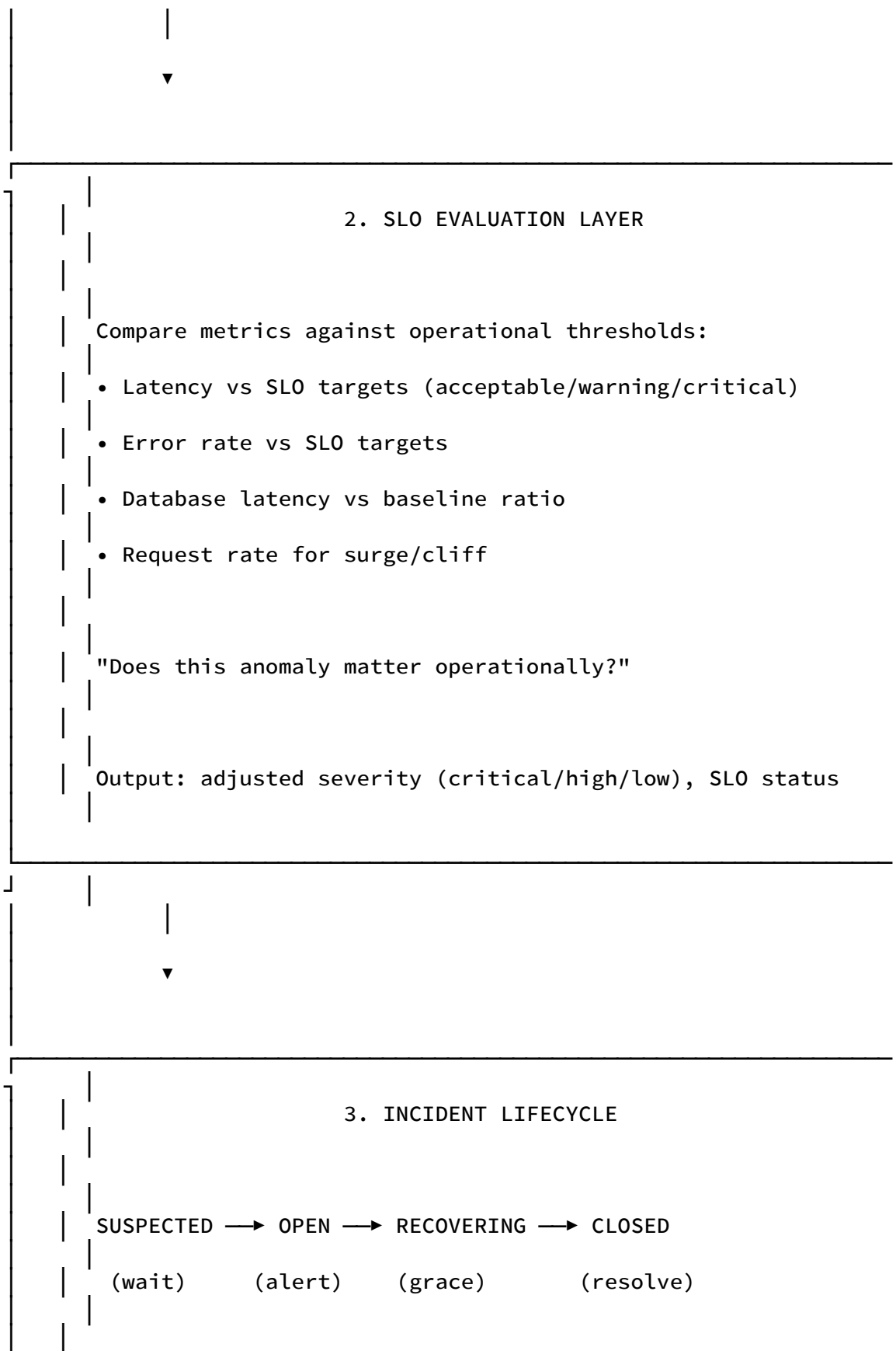
What is an Incident?

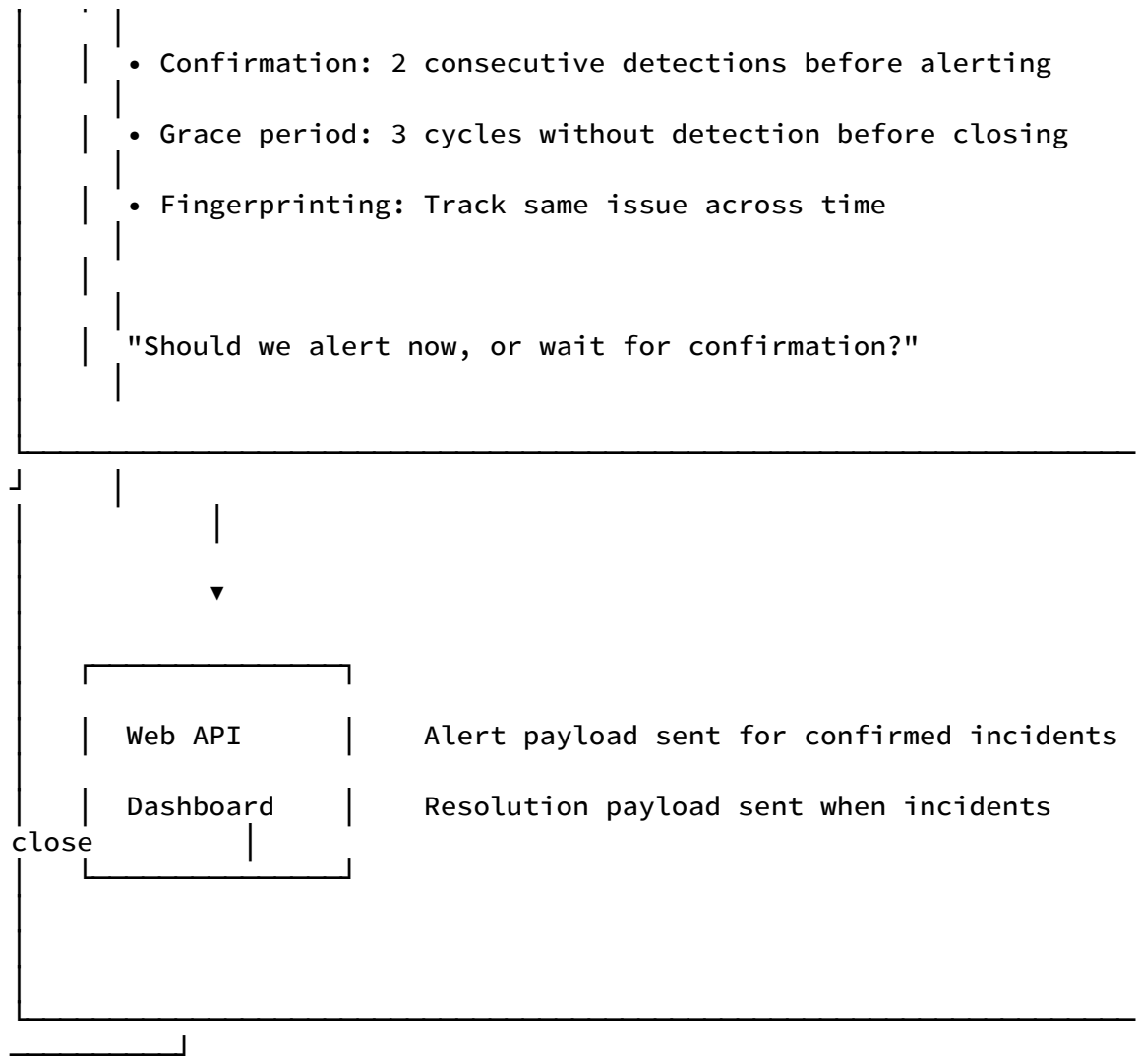
An **incident** is a tracked occurrence of an anomaly pattern. Yaga2 doesn't just detect anomalies—it tracks them over time to:

- **Confirm** anomalies aren't transient glitches (require 2+ consecutive detections)
- **Track duration** of ongoing issues
- **Prevent duplicates** by recognizing the same issue across detection cycles
- **Manage resolution** with grace periods to avoid flapping

System Architecture




2. SLO EVALUATION LAYER
Compare metrics against operational thresholds:
• Latency vs SLO targets (acceptable/warning/critical)
• Error rate vs SLO targets
• Database latency vs baseline ratio
• Request rate for surge/cliff
"Does this anomaly matter operationally?"
Output: adjusted severity (critical/high/low), SLO status
3. INCIDENT LIFECYCLE
SUSPECTED ⟶ OPEN ⟶ RECOVERING ⟶ CLOSED
(wait) (alert) (grace) (resolve)



Time-Aware Detection

Service behavior varies significantly by time:

Time Period	Typical Behavior
Business hours (Mon-Fri 8-18)	High traffic, tight latency requirements
Evening (Mon-Fri 18-22)	Moderate traffic
Night (22-06)	Low traffic, batch jobs
Weekend	Different patterns entirely

A 3 AM traffic level that would be alarming at 3 PM is completely normal at night. Yaga2 trains **separate models for each time period** to avoid false positives from expected behavioral differences.

The Five Metrics

Yaga2 monitors five core metrics for each service:

Metric	What It Measures	Why It Matters
request_rate	Requests per second	Traffic volume - sudden drops or spikes indicate problems
application_latency	Server processing time (ms)	User experience - slow responses frustrate users
dependency_latency	External dependency call time (ms)	Downstream issues - if a dependency is slow, you'll be slow
database_latency	Database query time (ms)	Database health - often the bottleneck
error_rate	Failed requests (0-1 ratio)	Reliability - errors directly impact users

How to Use This Guide

If You Want To...	Read...
Understand how ML detection works	Isolation Forest
Learn about named patterns	Pattern Matching
Configure SLO thresholds	SLO Evaluation
Understand alert timing	Incident Lifecycle

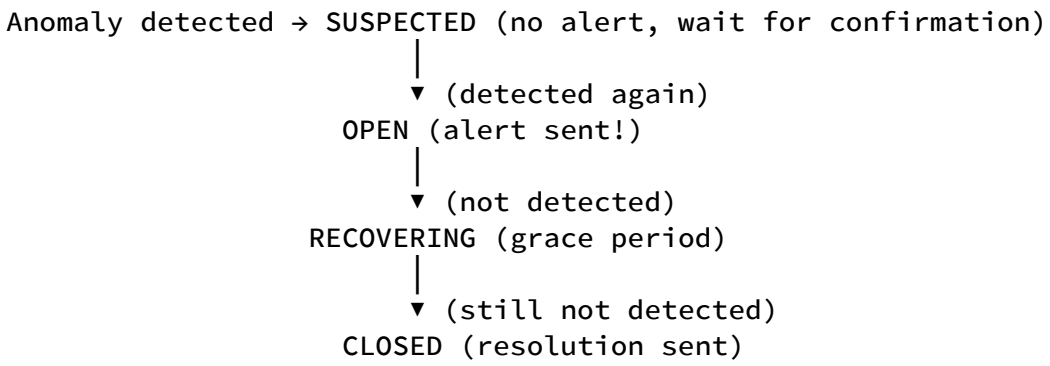
If You Want To...	Read...
Troubleshoot issues	Troubleshooting
Quick reference	Decision Matrix

Quick Reference

Severity Levels

Severity	Meaning	Action Required
critical	SLO breached, users impacted	Immediate response
high	Approaching SLO limits	Investigate promptly
low	Anomaly detected but within SLO	Monitor, no action

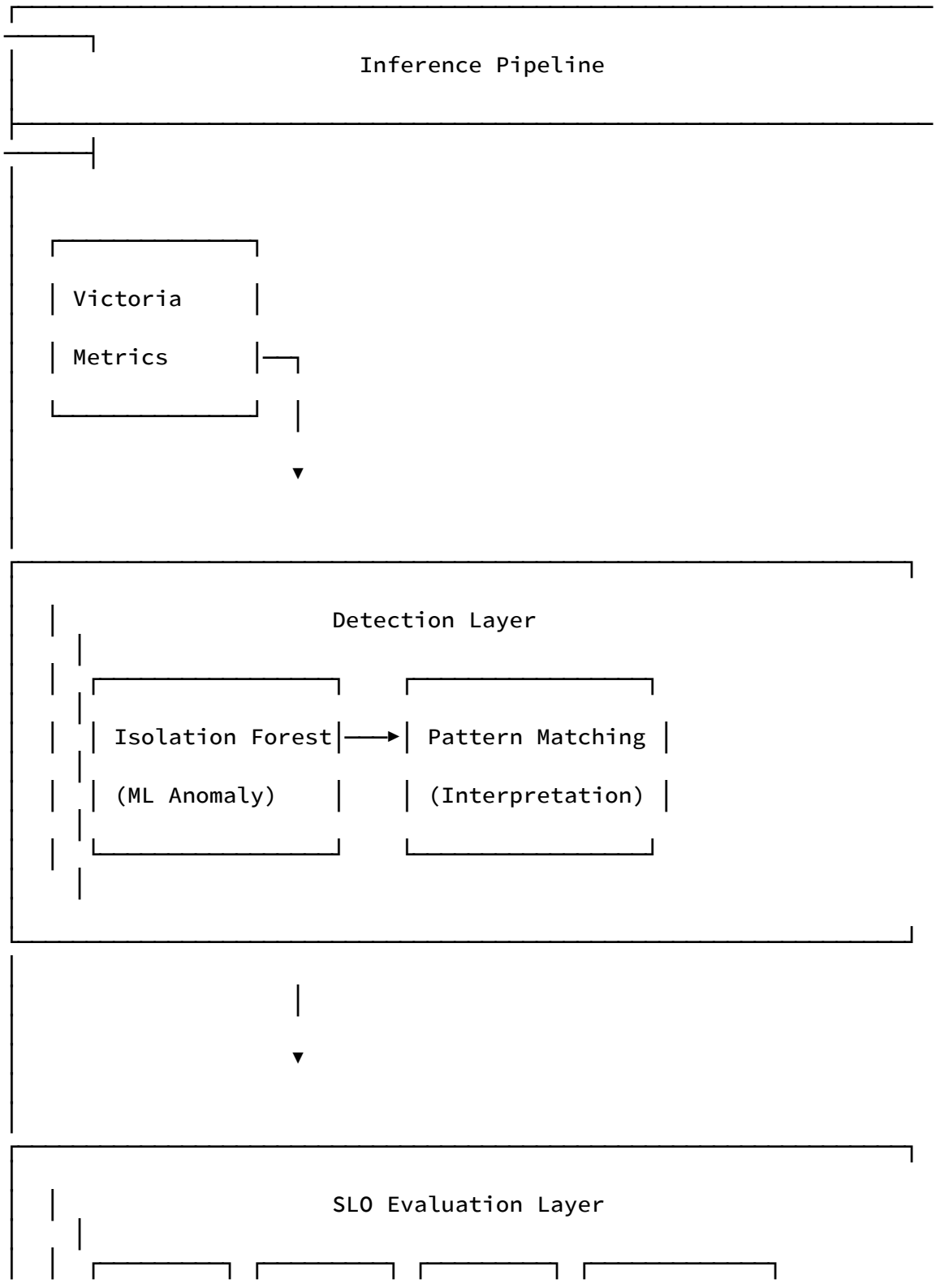
Alert Flow Summary

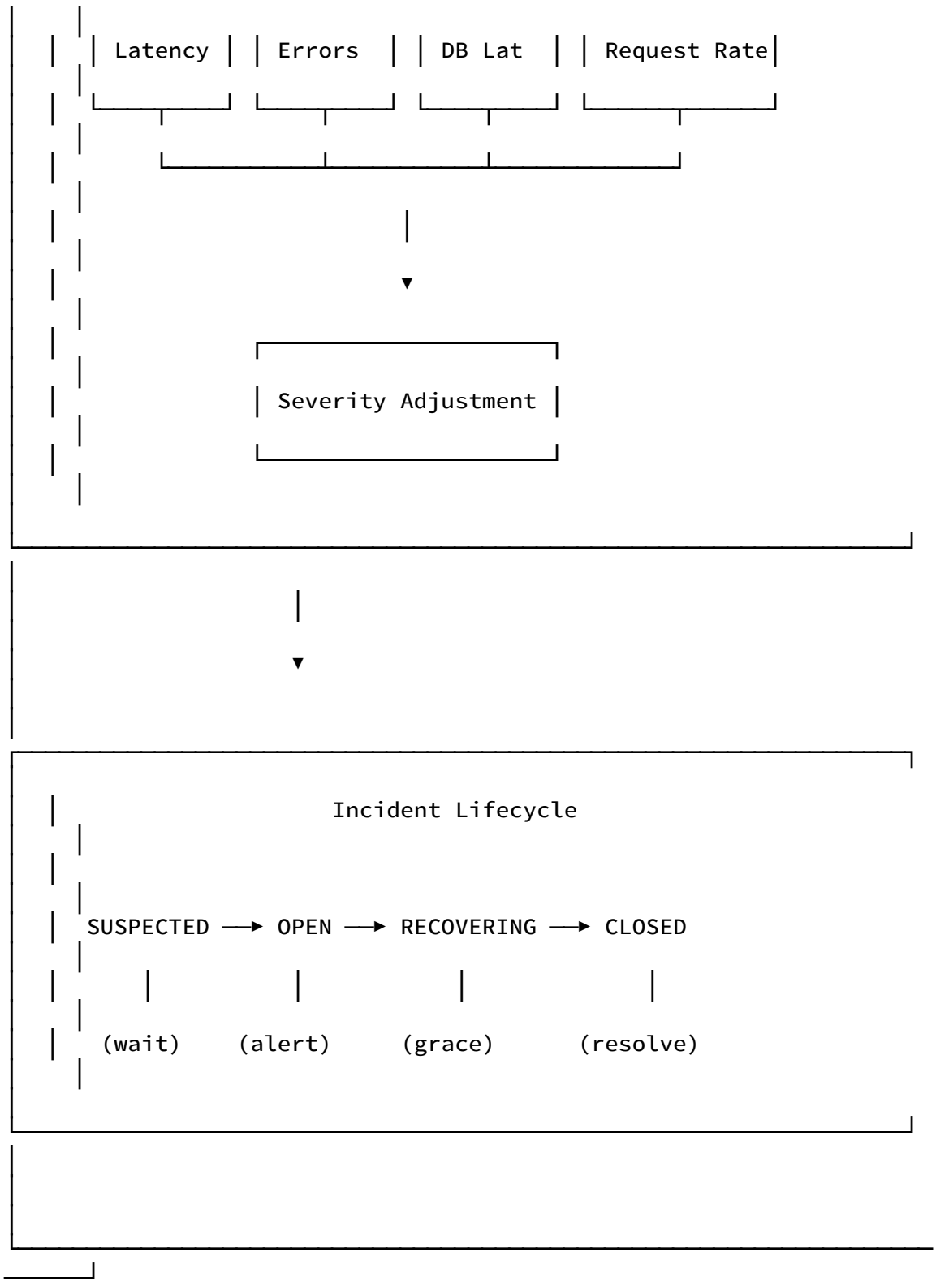


System Overview

The Yaga2 anomaly detection system processes metrics through multiple layers to produce actionable alerts.

Architecture




Latency
Errors
DB Lat
Request Rate
Severity Adjustment
Incident Lifecycle
SUSPECTED
OPEN
RECOVERING
CLOSED
(wait)
(alert)
(grace)
(resolve)

Metrics Collected

Metric	Description	Unit
`request_rate`	Incoming requests per second	req/s
`application_latency`	Server-side processing time	ms
`dependency_latency`	External dependency call time	ms
`database_latency`	Database query time	ms
`error_rate`	Percentage of failed requests	ratio (0-1)

Time-Aware Detection

The system uses **time-aware models** trained separately for each behavioral period:

Period	Hours	Days
`business_hours`	08:00 - 18:00	Mon-Fri
`evening_hours`	18:00 - 22:00	Mon-Fri
`night_hours`	22:00 - 06:00	Mon-Fri
`weekend_day`	08:00 - 22:00	Sat-Sun
`weekend_night`	22:00 - 08:00	Sat-Sun

This prevents false positives from expected behavioral differences (e.g., low traffic at 3 AM is normal).

Two-Pass Detection

For accurate cascade analysis, detection runs in two passes:

1. **Pass 1**: Detect anomalies for all services (no dependency context)
2. **Pass 2**: Re-analyze latency anomalies with dependency context

This enables identifying root cause services in dependency chains.

Output

Each inference run produces:

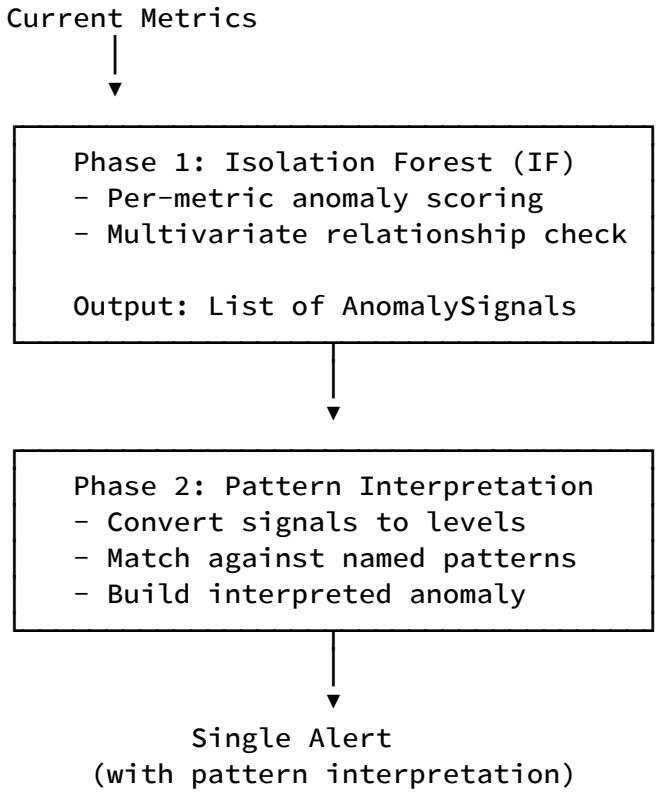
- **Anomaly alerts** for services with detected issues
- **Resolution notifications** for incidents that cleared
- **Enrichment data** (exceptions, service graph) when available

See API Payload for the complete output format.

Detection Layer

The detection layer identifies anomalies using a **sequential pipeline** where ML detection triggers pattern interpretation.

Pipeline Flow



Key Concepts

Anomaly Signal

When IF detects an anomaly, it produces a signal:

```
{
  "metric": "application_latency",
  "score": -0.35,
  "direction": "high",
  "percentile": 92.0
}
```

Metric Levels

Signals are converted to semantic levels for pattern matching:

Percentile	Level
> 95	`very_high`
90 - 95	`high`
80 - 90	`elevated`
70 - 80	`moderate`
10 - 70	`normal`
5 - 10	`low`
< 5	`very_low`

Lower-is-Better Metrics

Some metrics treat low values as improvements, not anomalies:

- `database_latency` - Faster queries are good
- `dependency_latency` - Faster dependencies are good
- `error_rate` - Fewer errors are good

For these metrics, when IF flags them as “low”, they are treated as `normal`.

Detection Methods

Method	Type	Best For
Isolation Forest	ML	Novel/unknown anomalies
Pattern Matching	Rule-based	Known operational scenarios

Sections

- Isolation Forest (ML) - How ML detection works
- Pattern Matching - Named patterns and interpretations
- Detection Pipeline - End-to-end detection flow

Isolation Forest (ML Detection)

Isolation Forest is an unsupervised machine learning algorithm that detects anomalies by measuring how easy it is to “isolate” a data point. This chapter explains how it works, why it’s effective for service metrics, and how Yaga2 uses it.

What is Unsupervised Learning?

Machine learning algorithms fall into two main categories:

- **Supervised learning**: Requires labeled training data. For anomaly detection, you’d need examples labeled “normal” and “anomaly.”
- **Unsupervised learning**: Learns patterns from unlabeled data. No need to manually identify anomalies in advance.

Why unsupervised matters for anomaly detection:

Labeling anomalies is impractical:

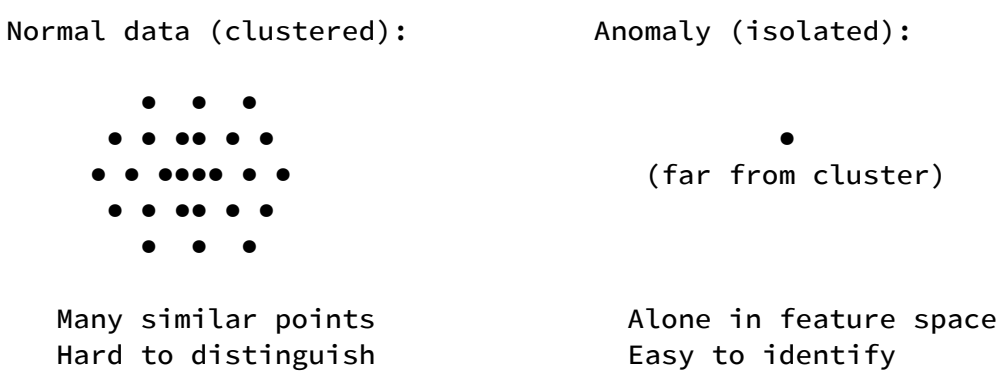
- You can’t predict all future failure modes
- What’s “anomalous” changes as services evolve
- Manual labeling is expensive and error-prone

Isolation Forest learns what’s “normal” from your historical data and flags anything that deviates—no labels required.

The Core Insight

Anomalies are easier to isolate than normal points.

Think about it visually:



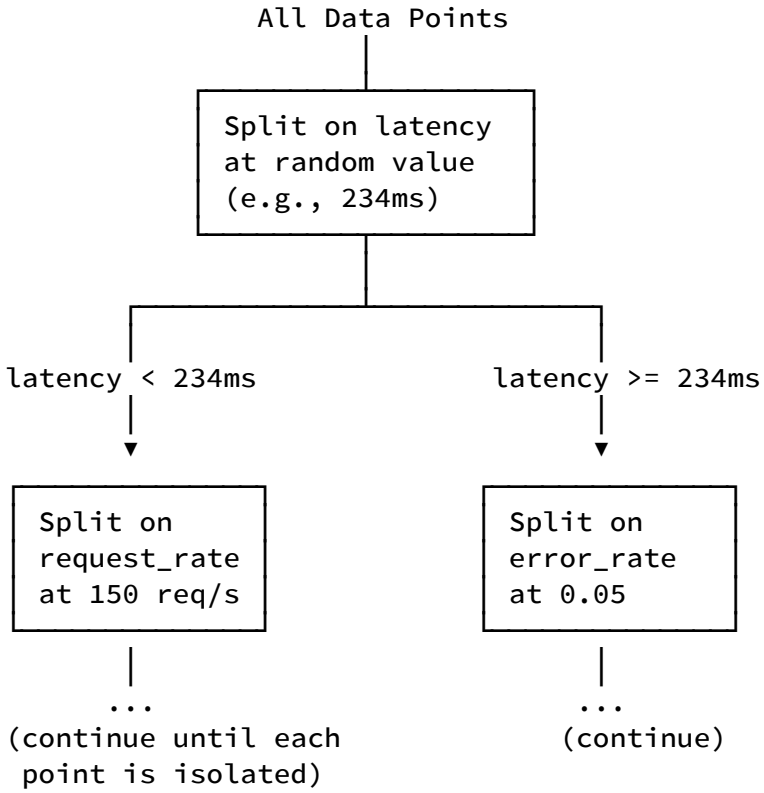
If you were to randomly draw dividing lines to separate data points:

- **Normal points**: Buried in a cluster, need many cuts to isolate
- **Anomalies**: Sitting alone, isolated in just 1-2 cuts

How Isolation Forest Works

Step 1: Build Random Trees (Training)

Isolation Forest builds many random decision trees (called “isolation trees”):



Each tree:

1. **Randomly picks a feature** (e.g., latency, error_rate)
2. **Randomly picks a split value** between the min and max of that feature
3. **Recursively partitions** until each point is isolated

Step 2: Measure Path Length

For each data point, count how many splits were needed to isolate it:

```
Anomaly path:                          Normal point path:

Root ⟶ Left ⟶ ISOLATED                Root ⟶ Right ⟶ Left ⟶ Right
⟶ Left ⟶ ISOLATED

Path length: 2                         Path length: 5
(suspicious!)                          (normal)
```

Key principle:

- Short paths → Anomalous (easy to isolate)
- Long paths → Normal (hard to isolate)

Step 3: Compute Anomaly Score

Average the path lengths across all trees and normalize:

$$\text{Anomaly Score} = \frac{2^{(-\text{average\_path\_length} / c(n))}}{\text{(normalized)}}$$

Where `c(n)` is the average path length of an unsuccessful search in a binary search tree of n samples.

Score interpretation:

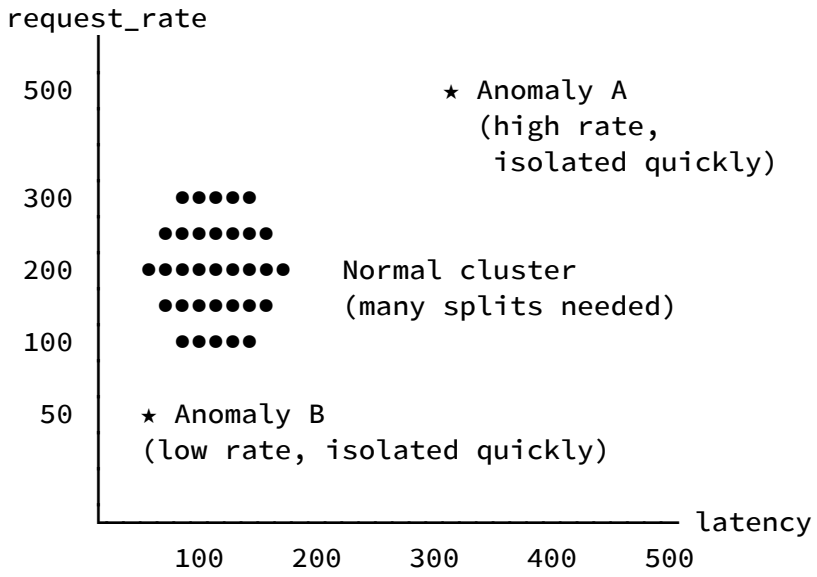
- **Score ≈ 1**: Highly anomalous (very short paths)
- **Score ≈ 0.5**: On the boundary
- **Score ≈ 0**: Normal (long paths)

Yaga2 uses the scikit-learn `decision_function()` which returns scores in [-1, 1]:

- **Negative scores**: Anomalous (more negative = more anomalous)
- **Positive scores**: Normal

Visual Example

Consider 2D data with latency and request_rate:




```
Random split 1: latency < 350ms
Random split 2: request_rate < 400

Result:
- Anomaly A: Isolated in 2 splits (score ≈ -0.5)
- Anomaly B: Isolated in 2 splits (score ≈ -0.5)
- Normal points: Need 5-8 splits (score ≈ 0.1)
```

Why Isolation Forest is Ideal for Service Metrics

Property	Why It Matters for Metrics
Unsupervised	No need to label historical incidents
Fast	O(n log n) - handles high-volume metrics
Handles skew	Latency distributions are typically long-tailed
No distribution assumptions	Unlike z-score, doesn't assume Gaussian
Works with multiple features	Can detect unusual metric *combinations*

Comparison with Other Methods

Method	Limitation for Metrics
Z-score / Standard Deviation	Assumes normal distribution - latency is skewed
Static thresholds	Brittle, need constant tuning
LSTM / Deep Learning	Requires labeled data, computationally expensive
Local Outlier Factor	Slow at inference time
One-Class SVM	Sensitive to hyperparameters

Model Types in Yaga2

Univariate Models (Per-Metric)

Separate model for each metric, detecting single-metric anomalies:

Model	What It Detects	Example
`request_rate_isolation`	Traffic anomalies	Sudden drop to 10% of normal
`application_latency_isolation`	Response time issues	Latency spike to 500ms (normally 100ms)
`error_rate_isolation`	Error spikes	Error rate jumps to 5% (normally 0.1%)
`dependency_latency_isolation`	Dependency slowness	External calls taking 2s

Model	What It Detects	Example
		(normally 200ms)
`database_latency_isolation`	Database issues	Query time spikes

Multivariate Model (Combined)

One model considering all metrics together. This detects unusual **combinations** that look normal individually:

```
Example: Fast-Fail Pattern

Individual metrics:
  request_rate: 100 req/s     ← Normal
  application_latency: 5ms    ← Very fast (good?)
  error_rate: 40%             ← High

Analysis:
  - Each metric might not trigger univariate detection
  - But the COMBINATION is suspicious:
    "Very fast responses + high errors = requests failing before
processing"

  Multivariate model catches this pattern!
```

Contamination Rate

Contamination is a key hyperparameter—the expected proportion of anomalies in training data.

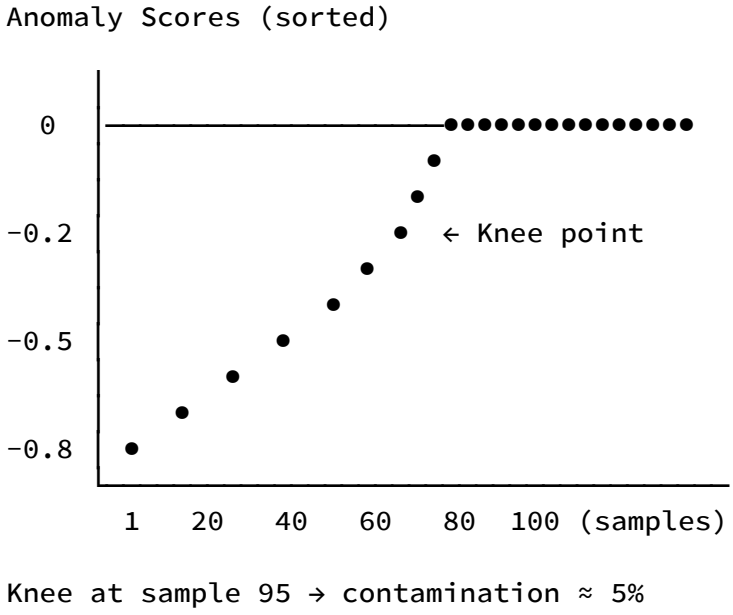
```
IsolationForest(contamination=0.05)  # Expect 5% anomalies
```

Contamination	Effect	Use Case
0.01-0.03	Very strict, few anomalies flagged	Critical services where false positives are costly
0.05	Balanced (default)	Most services
0.08-0.10	More sensitive, more anomalies flagged	Noisy services, development environments

How Yaga2 Sets Contamination

Yaga2 estimates contamination automatically using **knee detection**:

1. Train a preliminary model with `contamination="auto"`
2. Get anomaly scores for all training data
3. Sort scores and find the "knee" (bend in the curve)
4. Points beyond the knee are considered anomalies
5. Their proportion becomes the estimated contamination



Knee at sample 95 → contamination ≈ 5%

Severity Thresholds

Anomaly scores are mapped to severity levels:

Calibrated Thresholds (Preferred)

During training, thresholds are calibrated from validation data:

Severity	Percentile	Meaning
Critical	Bottom 0.1%	1 in 1000 - extremely rare
High	Bottom 1%	1 in 100 - significant
Medium	Bottom 5%	1 in 20 - moderate
Low	Bottom 10%	1 in 10 - minor

Fallback Thresholds

If calibration data unavailable:

Severity	Score Range
Critical	< -0.6
High	-0.6 to -0.3
Medium	-0.3 to -0.1
Low	>= -0.1

Training Process

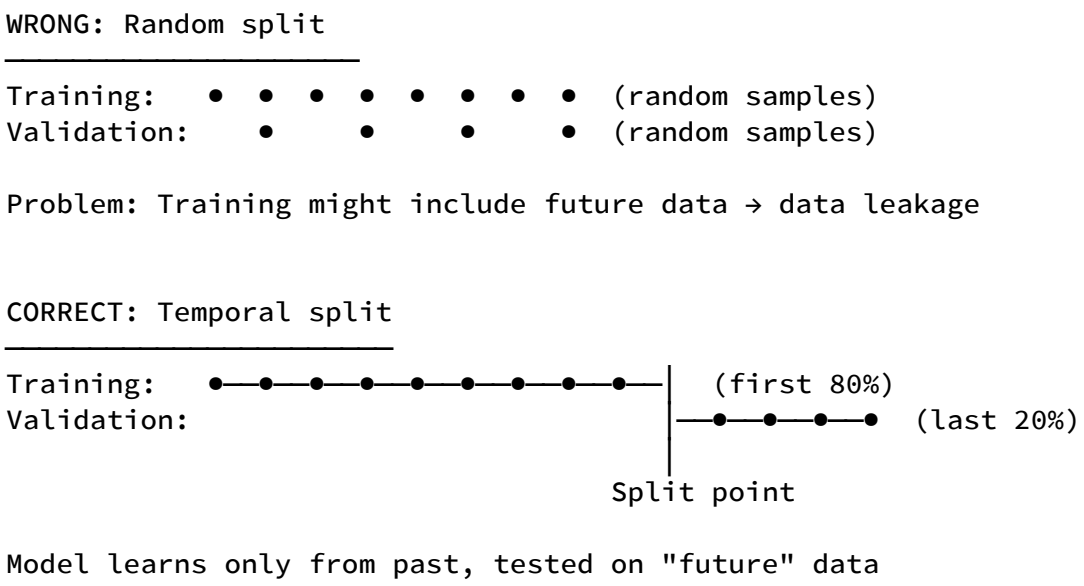
Data Requirements

Requirement	Value	Why
Training period	30 days	Capture weekly patterns
Min univariate samples	500	Statistical significance
Min multivariate samples	1000	Need more for cross-metric patterns
Data granularity	5 minutes	Balance detail vs noise

Training Steps

1. **Fetch 30 days of metrics** from VictoriaMetrics
2. **Segment by time period** (business hours, night, weekend)
3. **Temporal split**: 80% training, 20% validation (chronological)
4. **Estimate contamination** from data distribution
5. **Train univariate models** for each metric
6. **Train multivariate model** on all metrics
7. **Calibrate severity thresholds** on validation data
8. **Save models and metadata**

Why Temporal Split?



Time-Aware Models

Service behavior varies by time of day:

Time Period	Characteristics
business_hours (08-18 weekdays)	High traffic, tight latency
evening_hours (18-22 weekdays)	Moderate traffic
night_hours (22-06 weekdays)	Low traffic, batch jobs
weekend_day (08-22 Sat-Sun)	Variable patterns
weekend_night (22-08 Sat-Sun)	Very low traffic

Why separate models?

Without time awareness:

```
3 AM: 10 requests/second
  → Model: "This is way below the 500 req/s average - ANOMALY!"
  → Reality: This is normal at 3 AM
  → Result: False positive
```

With time-aware models:

```
3 AM: 10 requests/second
  → Night model average: 15 req/s
  → Model: "This is close to the night baseline - NORMAL"
  → Result: Correct!
```

Strengths and Limitations

Strengths

Strength	Explanation
No labeled data needed	Learns from normal patterns, no need to tag anomalies
Fast training and inference	O(n log n) complexity, scales to millions of points
Works with skewed data	Latency distributions are naturally long-tailed
Detects novel anomalies	Can find patterns never seen before
Handles multiple dimensions	Multivariate model catches unusual combinations

Limitations

Limitation	How Yaga2 Mitigates
No semantic understanding	Pattern matching interprets what anomalies mean
Context-blind	Time-aware models reduce false positives
Can produce false positives	SLO evaluation filters operationally insignificant anomalies
Training data quality matters	Data quality checks and minimum sample requirements
Gradual drift may be missed	Daily retraining, drift detection

How Detection Works at Inference Time

```
1. Current Metrics Arrive
   {request_rate: 450, latency: 280, errors: 0.02, ...}
                    |
                    ▼
2. Determine Time Period
   Current time: 10:30 AM Wednesday → "business_hours"
                    |
                    ▼
3. Load Appropriate Models
   booking/business_hours/model.joblib
                    |
                    ▼
4. Scale Metrics
   RobustScaler normalizes to training distribution
                    |
                    ▼
5. Score Each Model
   Univariate: latency_score = -0.35 (high!)
               request_score = 0.1 (normal)
               error_score = 0.05 (normal)
   Multivariate: combined_score = -0.28 (elevated)
                    |
                    ▼
6. Map to Severity
   latency_score -0.35 → severity: "high"
                    |
                    ▼
7. Pattern Matching
   High latency + normal traffic + normal errors
   → Pattern: "latency_spike_recent"
                    |
                    ▼
8. Output Anomaly
   {
     "pattern_name": "latency_spike_recent",
     "severity": "high",
     "score": -0.35,
     "description": "Latency spike: 280ms (normally 120ms)"
   }
```

Further Reading

- Pattern Matching - How anomalies are interpreted
- Detection Pipeline - End-to-end detection flow
- SLO Evaluation - How severity is adjusted

Pattern Matching

Pattern matching interprets ML signals by comparing metric combinations against known operational scenarios. This chapter explains what pattern matching is, how it works, and provides detailed examples of each pattern type.

What is Pattern Matching?

While Isolation Forest answers **“Is this unusual?”**, pattern matching answers **“What does this unusual behavior mean?”**

Pattern matching is a **rule-based interpretation layer** that recognizes known operational scenarios by examining how metrics relate to each other.

Why Pattern Matching?

ML detection tells you something is anomalous, but not what it means:

```
ML Output:
  application_latency: anomalous (score: -0.35)
  request_rate: anomalous (score: -0.28)
  error_rate: normal

Human question: "So... is this bad? What should I do?"
```

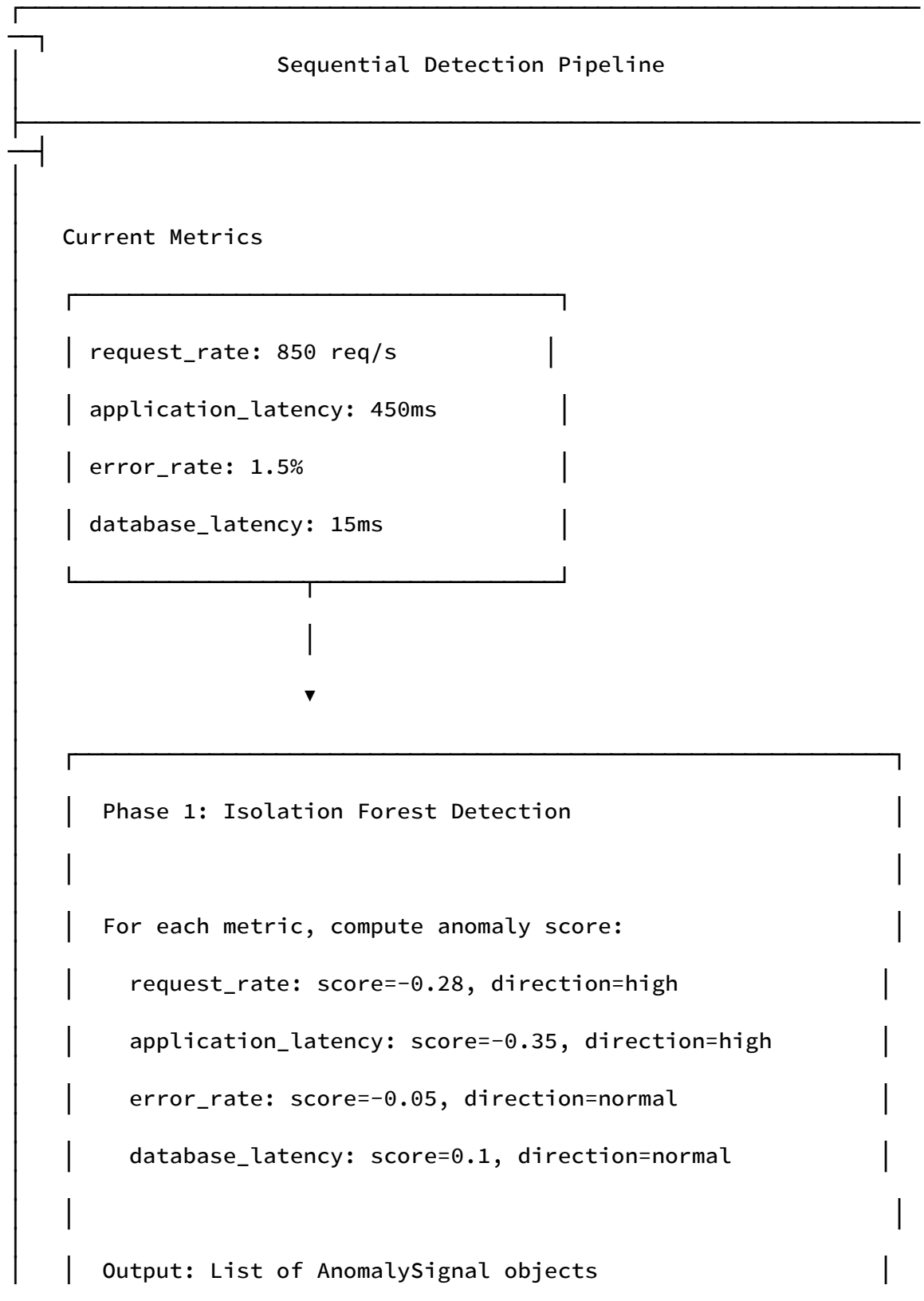
Pattern matching adds semantic meaning:

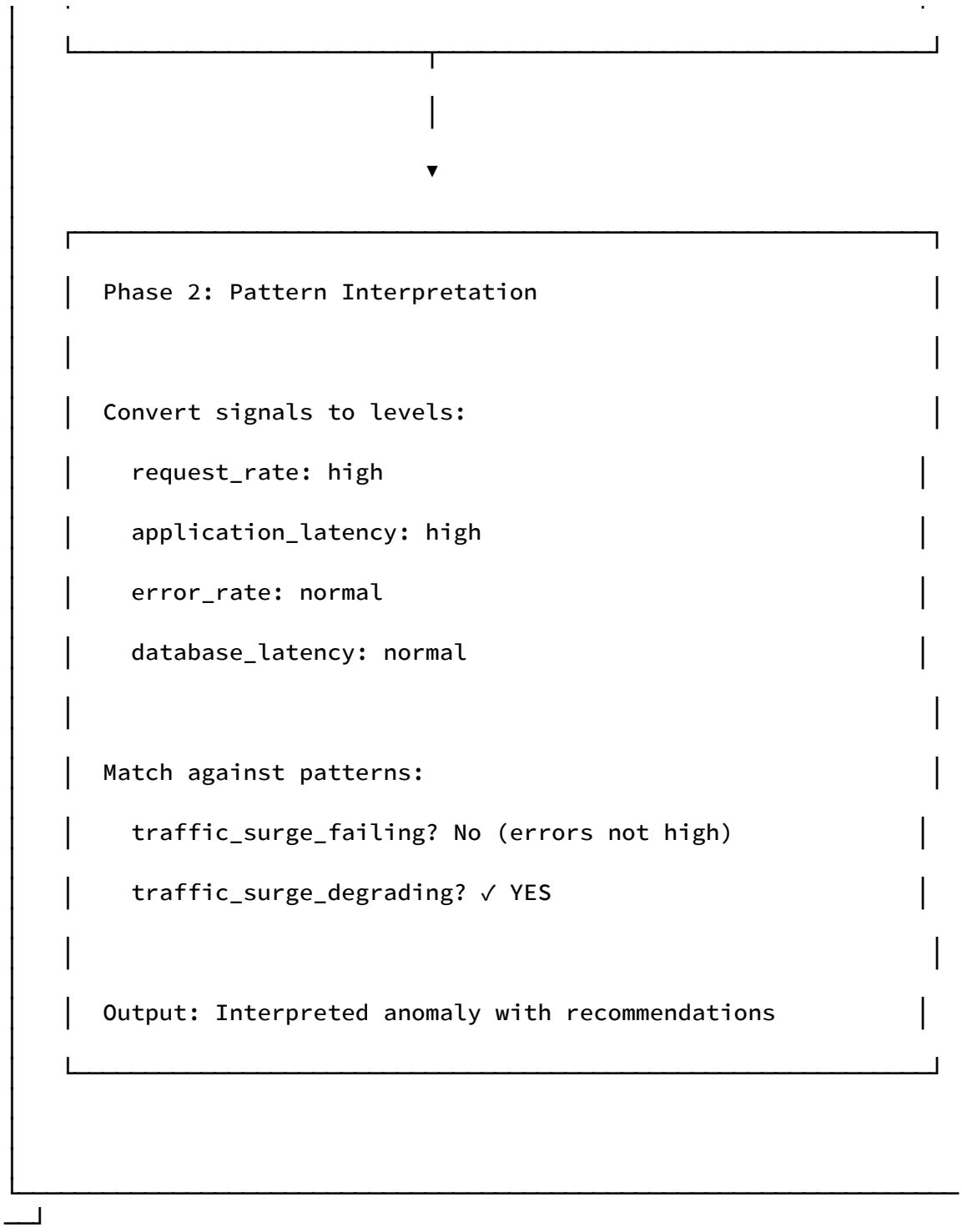
```
Pattern Match: traffic_surge_degrading

Interpretation: "You have a traffic surge (3x normal) that's causing
  latency degradation (450ms vs 150ms normal). Errors are still
normal,
  so you're approaching capacity but not failing yet."

Recommended Actions:
  1. Scale horizontally if possible
  2. Check resource bottlenecks (CPU, connections)
  3. Monitor error rate for signs of impending failure
```

ML vs Pattern Matching

Aspect	Isolation Forest	Pattern Matching
Approach	Statistical (unsupervised ML)	Rule-based (expert knowledge)
Question answered	“Is this unusual?”	“What does it mean?”
Learns from	Historical data	Domain expertise
Catches	Novel/unknown issues	Known operational scenarios
Output	Anomaly score	Named pattern + recommendations
Adapts	Automatically via training	Manually via pattern definitions

How They Work Together




Phase 2: Pattern Interpretation
Convert signals to levels:
request_rate: high
application_latency: high
error_rate: normal
database_latency: normal
Match against patterns:
traffic_surge_failing? No (errors not high)
traffic_surge_degrading? ✓ YES
Output: Interpreted anomaly with recommendations

How Pattern Matching Works

Step 1: Convert IF Signals to Metric Levels

Each metric is classified based on its Isolation Forest signal:

Percentile Range	Level
> 95th	`very_high`
90th - 95th	`high`
80th - 90th	`elevated`
70th - 80th	`moderate`
10th - 70th	`normal`
5th - 10th	`low`
< 5th	`very_low`
No IF signal	`normal`

Example:

```
IF says: latency is at 92nd percentile (score: -0.35)
Level: high (between 90th-95th)
```

Step 2: Handle "Lower is Better" Metrics

Some metrics are better when low:

Metric	Lower is Better?	Why
`error_rate`	Yes	0% errors is ideal
`database_latency`	Yes	Faster DB is better
`dependency_latency`	Yes	Faster dependencies is better

Metric	Lower is Better?	Why
`application_latency`	No*	Low + high errors = fast-fail

*Application latency is NOT in “lower is better” because low latency with high errors indicates fast failures (requests rejected before processing).

When IF flags a “lower is better” metric as anomalously low, it’s treated as `normal`:

```
IF says: database_latency is unusually low (faster than normal)
Pattern matching: Treat as "normal" (this is good, not a problem)
```

Step 3: Match Against Pattern Conditions

Each pattern has conditions that must be met:

```
"traffic_surge_degrading": {
    "conditions": {
        "request_rate": "high",           # Must be high
        "application_latency": "high",    # Must be high
        "error_rate": "normal",           # Must be normal
    }
}
```

The pattern matches only if **all conditions** are satisfied.

Step 4: Generate Interpreted Output

When a pattern matches, generate:

- Human-readable description
- Semantic interpretation
- Recommended actions
- Contributing metrics

Named Patterns - Detailed Reference

Traffic Patterns

These patterns relate to changes in traffic volume (request rate).

traffic_surge_healthy

PATTERN: traffic_surge_healthy

Conditions:

request_rate: high

application_latency: normal

error_rate: normal

Severity: low

What It Means:

Traffic has increased significantly (e.g., 3x normal) but the service is handling it gracefully. Latency and errors remain normal - your system has headroom.

Possible Causes:

- Marketing campaign launched
- Viral content / social media mention
- Seasonal peak (holiday shopping)
- Organic growth

Recommended Actions:

- Monitor for signs of degradation
- Verify traffic is legitimate (not bot/attack)
- Consider pre-emptive scaling if trend continues

Example Scenario:

Black Friday starts. Traffic jumps from 100 req/s to 350 req/s.

Latency stays at 150ms, errors at 0.1%.

System is handling the surge well.

traffic_surge_degrading

```
┌──────────────────────────────────────────────────────────────────────────┐
│ PATTERN: traffic_surge_degrading                                         │
├──────────────────────────────────────────────────────────────────────────┤
│                                                                          │
│ Conditions:                                                              │
│   request_rate: high                                                     │
│   application_latency: high                                              │
│   error_rate: normal                                                     │
│                                                                          │
│ Severity: high                                                           │
│                                                                          │
│ What It Means:                                                           │
│   Traffic surge is causing performance degradation. Users are            │
│   experiencing slower responses, but requests are still                  │
│   completing successfully. You're approaching capacity.                  │
│                                                                          │
│ Why Errors Are Still Normal:                                             │
│   The system is slowing down to cope with load rather than               │
│   failing. This is often due to:                                         │
│   • Thread pools filling up (requests queue)                             │
│   • Connection pools near exhaustion                                     │
│   • CPU at high utilization but not 100%                                 │
│                                                                          │
```

Recommended Actions:

- IMMEDIATE: Scale horizontally if possible
- CHECK: Resource utilization (CPU, memory, connections)
- MONITOR: Error rate for signs of impending failure
- CONSIDER: Enable request throttling to protect backend

Example Scenario:

Traffic: 850 req/s (normally 200 req/s)

Latency: 450ms (normally 120ms)

Errors: 0.5% (normal)

Users notice slowness but can still complete transactions.

traffic_surge_failing

```
┌──────────────────────────────────────────────────────────────────────────────┐
│ PATTERN: traffic_surge_failing                                               │
├──────────────────────────────────────────────────────────────────────────────┤
│                                                                              │
│ Conditions:                                                                  │
│   request_rate: high                                                         │
│   application_latency: high                                                  │
│   error_rate: high                                                           │
│                                                                              │
│ Severity: critical                                                           │
│                                                                              │
│ What It Means:                                                               │
│   Service is at or beyond capacity. The traffic surge has                    │
│   overwhelmed the system - both latency and errors are elevated.             │
│   Users are actively being impacted.                                         │
│                                                                              │
│ Why This Is Critical:                                                        │
│   High errors + high latency = users are both waiting AND                    │
│   failing. This is the worst user experience - slow failures.                │
│                                                                              │
│ Recommended Actions:                                                         │
│   • IMMEDIATE: Scale horizontally or add capacity                            │
│   • IMMEDIATE: Enable rate limiting to protect backend                       │
```

- INVESTIGATE: Confirm traffic is legitimate vs attack
- PREPARE: Have rollback ready if recent deployment caused it

Example Scenario:

Traffic: 1200 req/s (normally 200 req/s)

Latency: 2500ms (normally 120ms)

Errors: 15%

System is collapsing under load. Many users seeing errors, others waiting forever for timeouts.

traffic_cliff

```
┌──────────────────────────────────────────────────────────────────────┐
│ PATTERN: traffic_cliff                                               │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│ Conditions:                                                          │
│   request_rate: very_low (sudden drop to <10% of normal)             │
│                                                                      │
│ Severity: critical                                                   │
│                                                                      │
│ What It Means:                                                       │
│   Traffic has suddenly dropped to near-zero. This often              │
│   indicates an upstream problem preventing requests from             │
│   reaching the service.                                              │
│                                                                      │
│ Common Causes:                                                       │
│   • DNS failure (users can't resolve hostname)                       │
│   • Load balancer misconfigured (routing to wrong backend)           │
│   • Firewall rule blocking traffic                                   │
│   • CDN/edge failure                                                 │
│   • Upstream service failure                                         │
│   • Network partition                                                │
│                                                                      │
│ Why Not Low Latency Pattern:                                         │
```

```
  The few requests getting through might show normal latency,
  but the problem is that NO requests are arriving.


Recommended Actions:
  • IMMEDIATE: Check upstream services and load balancers
  • CHECK: DNS resolution from multiple locations
  • VERIFY: Network connectivity and firewall rules
  • CORRELATE: Check if other services are also affected


Example Scenario:
  Traffic: 5 req/s (normally 200 req/s) - 97.5% drop

  At 2:15 PM, traffic suddenly drops from 200 req/s to 5 req/s.
  Investigation reveals DNS TTL expired and DNS provider had
  an outage, causing resolution failures.
```

Error Patterns

These patterns relate to error rate anomalies.

error_rate_elevated

```
┌──────────────────────────────────────────────────────────────────────┐
│ PATTERN: error_rate_elevated                                         │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│ Conditions:                                                          │
│   request_rate: normal                                               │
│   application_latency: normal                                        │
│   error_rate: high                                                   │
│                                                                      │
│ Severity: high                                                       │
│                                                                      │
│ What It Means:                                                       │
│   Error rate has increased significantly, but traffic and            │
│   latency are normal. This usually indicates a specific code         │
│   path or endpoint is failing, not a systemic issue.                 │
│                                                                      │
│ Common Causes:                                                       │
│   • Bug in a specific endpoint                                       │
│   • External dependency failing for certain operations               │
│   • Bad input data causing validation failures                       │
│   • Feature flag enabled broken code                                 │
│   • Database constraint violations                                   │
│                                                                      │
```

Why Latency Is Normal:

Most requests succeed normally - only a subset is failing.

The average latency doesn't change much because successes

dominate the metrics.

Recommended Actions:

- INVESTIGATE: Check exception types in exception_context
- CHECK: Recent deployments or configuration changes
- IDENTIFY: Which endpoints/operations are failing
- VERIFY: External dependency health

Example Scenario:

Traffic: 150 req/s (normal)

Latency: 120ms (normal)

Errors: 3.5% (normally 0.1%)

Exception breakdown shows 85% of errors are

"PaymentGatewayException" - the payment provider is having

issues, but only checkout flow is affected.

error_rate_critical

```
┌──────────────────────────────────────────────────────────────────────────┐
│ PATTERN: error_rate_critical                                             │
├──────────────────────────────────────────────────────────────────────────┤
│                                                                          │
│ Conditions:                                                              │
│   request_rate: normal                                                   │
│   application_latency: normal                                            │
│   error_rate: very_high (>5%)                                            │
│                                                                          │
│ Severity: critical                                                       │
│                                                                          │
│ What It Means:                                                           │
│   A significant portion of requests are failing. While the               │
│   service responds quickly (suggesting infrastructure is OK),            │
│   business logic is failing at a high rate.                              │
│                                                                          │
│ Common Causes:                                                           │
│   • Critical bug in recent deployment                                    │
│   • Database schema mismatch after migration                             │
│   • External API contract changed                                        │
│   • Authentication/authorization failure                                 │
│   • Missing configuration or secrets                                     │
│                                                                          │
```

```
Recommended Actions:
  • IMMEDIATE: Check recent deployments (rollback candidate)
  • INVESTIGATE: Exception breakdown for root cause
  • CHECK: Database connectivity and schema
  • VERIFY: All required configuration/secrets present

Example Scenario:
  Traffic: 150 req/s (normal)
  Latency: 80ms (normal - fast errors)
  Errors: 25%

  Exception breakdown: 100% "DatabaseSchemaException"
  Recent deployment added a required column, but migration
  didn't run in production.
```

Latency Patterns

These patterns relate to response time anomalies.

latency_spike_recent

```
┌──────────────────────────────────────────────────────────────────────────┐
│ PATTERN: latency_spike_recent                                            │
├──────────────────────────────────────────────────────────────────────────┤
│                                                                          │
│ Conditions:                                                              │
│   request_rate: normal                                                   │
│   application_latency: high                                              │
│   error_rate: normal                                                     │
│                                                                          │
│ Severity: high                                                           │
│                                                                          │
│ What It Means:                                                           │
│   Latency increased without traffic change. Something changed            │
│   in the system - this is NOT a capacity issue (traffic is               │
│   normal). Likely a recent deployment, config change, or                 │
│   dependency degradation.                                                │
│                                                                          │
│ Why Traffic Being Normal Matters:                                        │
│   If traffic were high, latency increase would make sense                │
│   (capacity). With normal traffic, the slowdown is internal.             │
│                                                                          │
│ Common Causes:                                                           │
│   • Recent deployment introduced inefficiency                            │
```

- Database query plan regression
- New logging/tracing overhead
- Downstream service slower
- GC pressure from memory leak
- Missing cache (cold cache after restart)

Recommended Actions:

- IMMEDIATE: Check deployments in last 2 hours
- CHECK: Configuration changes (feature flags, settings)
- CHECK: External dependency response times
- CHECK: Database query performance
- CHECK: GC behavior and memory pressure

Example Scenario:

Traffic: 150 req/s (normal)

Latency: 450ms (normally 150ms)

Errors: 0.1% (normal)

Timeline shows latency jumped at 10:30 AM.

Deployment log shows release v2.3.4 deployed at 10:28 AM.

New release added detailed audit logging that's synchronous.

internal_latency_issue

```
┌──────────────────────────────────────────────────────────────────────────┐
│ PATTERN: internal_latency_issue                                          │
├──────────────────────────────────────────────────────────────────────────┤
│                                                                          │
│ Conditions:                                                              │
│   application_latency: high                                              │
│   dependency_latency: normal                                             │
│   database_latency: normal                                               │
│                                                                          │
│ Severity: high                                                           │
│                                                                          │
│ What It Means:                                                           │
│   The service itself is slow, but all its dependencies                   │
│   (database, external services) are fast. The problem is                 │
│   INTERNAL to the application code.                                      │
│                                                                          │
│ Why This Is Significant:                                                 │
│   Eliminates external causes. The fix must be in this service,           │
│   not upstream/downstream.                                               │
│                                                                          │
│ Common Causes:                                                           │
│   • CPU-intensive computation                                            │
│   • Memory allocation/GC pressure                                        │
```

- Lock contention / thread starvation
- Inefficient algorithm ($O(n^2)$ loop)
- Synchronous I/O blocking event loop
- Missing async/await causing serialization

Recommended Actions:

- PROFILE: CPU profiling to find hot paths
- CHECK: Thread dumps for lock contention
- CHECK: Memory usage and GC logs
- REVIEW: Recent code changes for algorithmic issues

Example Scenario:

App latency: 800ms

Client latency: 50ms (fast external calls)

DB latency: 10ms (fast queries)

Where's the extra 740ms? Profiling reveals a nested loop iterating over large collections - $O(n^2)$ complexity.

database_bottleneck

```
┌──────────────────────────────────────────────────────────────────────────┐
│ PATTERN: database_bottleneck                                             │
├──────────────────────────────────────────────────────────────────────────┤
│                                                                          │
│ Conditions:                                                              │
│   database_latency: high                                                 │
│   application_latency: high                                              │
│   db_latency_ratio: > 0.7 (DB is >70% of total latency)                  │
│                                                                          │
│ Severity: high                                                           │
│                                                                          │
│ What It Means:                                                           │
│   Database operations dominate response time. The database               │
│   is the bottleneck - fixing app code won't help much.                   │
│                                                                          │
│ Why The Ratio Matters:                                                   │
│   If DB is 15ms out of 500ms total, DB isn't the problem.                │
│   If DB is 400ms out of 500ms total, DB IS the problem.                  │
│                                                                          │
│ Common Causes:                                                           │
│   • Missing database index                                               │
│   • Query plan regression (statistics stale)                             │
│   • Lock contention in database                                          │
```

- Database server under-provisioned
- N+1 query pattern
- Full table scans

Recommended Actions:

- INVESTIGATE: Slow query logs
- CHECK: Missing indexes on frequently queried columns
- CHECK: Database CPU and I/O metrics
- ANALYZE: Query execution plans
- CONSIDER: Query caching or read replicas

Example Scenario:

App latency: 650ms

DB latency: 520ms (80% of total)

Client latency: 10ms

Slow query log shows: SELECT * FROM orders WHERE customer_id = ? taking 500ms. Missing index on customer_id.

downstream_cascade

```
┌──────────────────────────────────────────────────────────────────────────────┐
│  PATTERN: downstream_cascade                                                 │
├──────────────────────────────────────────────────────────────────────────────┤
│                                                                              │
│  Conditions:                                                                 │
│    dependency_latency: high                                                  │
│    application_latency: high                                                 │
│    dependency_latency_ratio: > 0.6 (external calls >60% of total)            │
│                                                                              │
│  Severity: high                                                              │
│                                                                              │
│  What It Means:                                                              │
│    External dependency calls are dominating response time.                   │
│    The service you're looking at is slow BECAUSE something                   │
│    it depends on is slow.                                                    │
│                                                                              │
│  Why This Pattern Exists:                                                    │
│    Helps you avoid blaming the wrong service. If booking                     │
│    is slow because search is slow, fix search, not booking.                  │
│                                                                              │
│  What To Look At:                                                            │
│    • service_graph_context shows downstream call breakdown                   │
│    • Identify which downstream service is slowest                            │
```

- That service is likely the root cause

Recommended Actions:

- FOCUS: Investigate the slow downstream service
- CHECK: service_graph_context for detailed breakdown
- CORRELATE: Does downstream service also have anomalies?
- CONSIDER: Circuit breaker to fail fast

Example Scenario:

App latency: 1200ms

Client latency: 950ms (79% of total)

DB latency: 15ms

service_graph_context shows:

search-api: 850ms avg (highest)

payment-api: 50ms avg

inventory-api: 40ms avg

Root cause: search-api is slow, causing booking to be slow.

Fast-Fail Patterns

These patterns indicate requests failing before normal processing.

fast_rejection

```
┌──────────────────────────────────────────────────────────────────────────────┐
│ PATTERN: fast_rejection                                                      │
├──────────────────────────────────────────────────────────────────────────────┤
│                                                                              │
│ Conditions:                                                                  │
│   application_latency: very_low                                              │
│   error_rate: very_high                                                      │
│                                                                              │
│ Severity: critical                                                           │
│                                                                              │
│ What It Means:                                                               │
│   Requests are failing VERY fast. They're being rejected                     │
│   before reaching normal application logic. Something is                     │
│   blocking requests at the entry point.                                      │
│                                                                              │
│ Why Low Latency + High Errors Is Bad:                                        │
│   Normal errors take time (processing, then failure).                        │
│   Fast errors mean immediate rejection - no processing at all.               │
│                                                                              │
│ Common Causes:                                                               │
│   • Circuit breaker OPEN (rejecting all requests)                            │
│   • Rate limiter active (over quota)                                         │
│   • Authentication/authorization failure                                     │
```

- Invalid API key or expired token
- TLS certificate mismatch
- Connection pool exhausted (immediate rejection)

Recommended Actions:

- IMMEDIATE: Check circuit breaker status
- CHECK: Rate limiter configuration and current rate
- CHECK: Authentication service health
- VERIFY: API keys and certificates are valid

Example Scenario:

Latency: 5ms (normally 150ms)

Errors: 95%

5ms is just enough time to check auth and reject.

Auth service is down, causing all requests to fail at

the middleware layer before reaching business logic.

fast_failure

```
┌──────────────────────────────────────────────────────────────────────────────┐
│ PATTERN: fast_failure                                                        │
├──────────────────────────────────────────────────────────────────────────────┤
│                                                                              │
│ Conditions:                                                                  │
│   application_latency: low                                                   │
│   error_rate: high                                                           │
│                                                                              │
│ Severity: critical                                                           │
│                                                                              │
│ What It Means:                                                               │
│   Requests are failing quickly - faster than normal processing               │
│   would take. Similar to fast_rejection but with slightly                    │
│   more processing time (maybe some validation runs).                         │
│                                                                              │
│ The Latency-Error Relationship:                                              │
│   Normal request: Validate → Process → DB → External → Return                │
│   Fast failure: Validate → Fail (skip processing)                            │
│                                                                              │
│ Common Causes:                                                               │
│   • Early validation failure                                                 │
│   • Feature flag disabled critical path                                      │
│   • Missing required configuration                                           │
```

```
  • Database connection failure (fail after connect attempt)


Recommended Actions:

  • CHECK: Application logs for error patterns

  • CHECK: Required configuration and feature flags

  • CHECK: Database and cache connectivity

  • REVIEW: Recent deployments for breaking changes


Example Scenario:

  Latency: 25ms (normally 200ms)

  Errors: 80%


  Logs show: "Redis connection failed"

  Redis cluster is unreachable. Service tries to connect

  (25ms timeout), fails, returns error.
```

Cascade Patterns

These patterns relate to dependency failures propagating.

upstream_cascade

```
┌──────────────────────────────────────────────────────────────────────────────┐
│ PATTERN: upstream_cascade                                                    │
├──────────────────────────────────────────────────────────────────────────────┤
│                                                                              │
│ Conditions:                                                                  │
│   application_latency: high                                                  │
│   _dependency_context: upstream_anomaly                                      │
│                                                                              │
│ Severity: high                                                               │
│                                                                              │
│ What It Means:                                                               │
│   This service is slow, AND at least one of its upstream                     │
│   dependencies also has an anomaly. The root cause is likely                 │
│   in the upstream service, not here.                                         │
│                                                                              │
│ How Cascade Detection Works:                                                 │
│   1. Yaga2 runs detection for all services (Pass 1)                          │
│   2. For services with latency anomalies, check dependencies                 │
│   3. If a dependency has an anomaly, it's a cascade                          │
│   4. Trace the chain to find the root cause service                          │
│                                                                              │
│ Why This Matters:                                                            │
│   Without cascade detection, you'd get 5 alerts for 5 services               │
```

when only 1 service (the root cause) needs fixing.

Example Dependency Chain:

mobile-api → booking → vms → titan

If titan has a database issue, all 4 services will be slow.

Cascade detection identifies titan as root cause.

cascade_analysis Output:

```
{
  "is_cascade": true,
  "root_cause_service": "titan",
  "affected_chain": ["titan", "vms", "booking", "mobile-api"],
  "cascade_type": "upstream_cascade"
}
```

Recommended Actions:

- FOCUS: Investigate root_cause_service first
- IGNORE: Don't fix downstream services (they'll recover)
- CORRELATE: Check root cause service's anomaly for details

Pattern Matching in Action

Complete Example: Traffic Surge

```
Input Metrics at 2:30 PM:
──────────────────────────────────────────────────────────────

  request_rate: 850 req/s (baseline: 200 req/s)
  application_latency: 420ms (baseline: 120ms)
  error_rate: 0.5% (baseline: 0.1%)
  database_latency: 15ms (baseline: 12ms)
  dependency_latency: 45ms (baseline: 40ms)

Phase 1: Isolation Forest
──────────────────────────────────────────────────────────────

  request_rate:        score=-0.42  percentile=96  → very_high
  application_latency: score=-0.35  percentile=94  → high
  error_rate:          score=-0.08  percentile=72  → normal
  database_latency:    score=0.05   percentile=55  → normal
  dependency_latency:      score=0.02   percentile=52  → normal

Phase 2: Level Classification
──────────────────────────────────────────────────────────────

  request_rate: very_high (>95th percentile)
  application_latency: high (90-95th percentile)
  error_rate: normal (errors not anomalous)
  database_latency: normal (not flagged by IF)
  dependency_latency: normal (not flagged by IF)

Phase 3: Pattern Matching
──────────────────────────────────────────────────────────────

  Checking: traffic_surge_failing
    Conditions: request_rate=high ✓, latency=high ✓, errors=high ✗
    Result: NO MATCH (errors are normal)

  Checking: traffic_surge_degrading
    Conditions: request_rate=high ✓, latency=high ✓, errors=normal ✓
    Result: MATCH ✓

Output:
──────────────────────────────────────────────────────────────

{
  "pattern_name": "traffic_surge_degrading",
  "severity": "high",
  "description": "Traffic surge causing slowdown: 850 req/s driving
```

```
    latency to 420ms (errors stable at 0.50%)",
  "interpretation": "Service is slowing under load but not failing -
    approaching capacity. Users experiencing degraded performance
but
    requests are completing.",
  "recommended_actions": [
    "IMMEDIATE: Scale horizontally if possible",
    "CHECK: Resource bottlenecks (CPU, memory, connections)",
    "MONITOR: Error rate for signs of impending failure",
    "CONSIDER: Enable request throttling to protect backend"
  ],
  "metric_levels": {
    "request_rate": "very_high",
    "application_latency": "high",
    "error_rate": "normal"
  }
}
```

Adding New Patterns

To add a new pattern, edit `smartbox_anomaly/detection/interpretations.py`:

```
MULTIVARIATE_PATTERNS["cache_miss_storm"] = PatternDefinition(
    name="cache_miss_storm",
    conditions={
        "request_rate": "normal",
        "database_latency": "high",
        "db_latency_ratio": "> 0.8",
    },
    severity="high",
    message_template=(
        "Cache miss storm: {database_latency:.0f}ms DB latency "
        "({db_latency_ratio:.0%} of total response time)"
    ),
    interpretation=(
        "Database is handling requests that should be cached. "
        "Cache service may be down or TTL expired for hot keys."
    ),
    recommended_actions=[
        "CHECK: Cache service health (Redis/Memcached)",
        "CHECK: Cache hit rate metrics",
        "VERIFY: Cache TTL hasn't expired for hot keys",
        "CONSIDER: Implementing cache warming",
    ],
)
```

Pattern Conditions

Condition Type	Example	Description
Level	`"request_rate": "high"`	Metric must be at this level
Ratio	`"db_latency_ratio": "> 0.7"`	Ratio must exceed threshold
Any	`"error_rate": "any"`	Matches any level
Dependency	`"_dependency_context": "upstream_anomaly"`	Cascade detection

Recommendation Prefixes

Use these prefixes for consistent action ordering:

Prefix	Priority	Use For
`IMMEDIATE`	1	Actions needed right now
`CHECK`	2	Things to investigate
`INVESTIGATE`	3	Deeper analysis needed
`MONITOR`	4	Ongoing observation
`CONSIDER`	5	Optional improvements

Further Reading

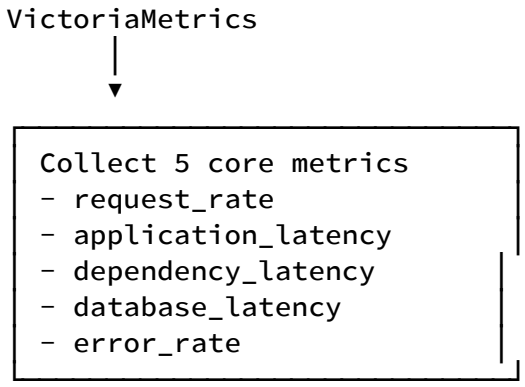
- Isolation Forest - How ML detection works
- Detection Pipeline - End-to-end detection flow
- API Payload Reference - Output format

Detection Pipeline

The complete detection pipeline from metrics collection to anomaly output.

Pipeline Phases

Phase 1: Metrics Collection



Phase 2: Input Validation

Metrics are validated before processing:

Check	Action
NaN/Inf values	Replaced with 0.0
Negative rates	Capped at 0.0
Negative latencies	Capped at 0.0
Extreme latencies (>5 min)	Capped at 300,000ms
Extreme request rates (>1M/s)	Capped at 1,000,000
Error rates > 100%	Capped at 1.0

Validation warnings are included in the output.

Phase 3: Pass 1 Detection

First pass detects anomalies without dependency context:

```
For each service:
    1. Load time-aware model (based on current time period)
    2. Run univariate IF on each metric
    3. Run multivariate IF on combined metrics
    4. Collect anomaly signals
    5. Match signals against patterns
    6. Store result
```

Phase 4: Pass 2 Detection (Cascade Analysis)

Second pass re-analyzes services with latency anomalies:

```
For each service with latency anomaly:
    1. Build dependency context from Pass 1 results
    2. Check upstream dependencies for anomalies
    3. Re-run pattern matching with dependency info
    4. Add cascade_analysis if root cause found
```

Dependency Context:

```
{
  "upstream_status": "anomaly",
  "dependencies": {
    "vms": {"has_anomaly": true, "type": "database_bottleneck"},
    "titan": {"has_anomaly": true, "type": "latency_elevated"}
  },
  "root_cause_service": "titan"
}
```

Phase 5: SLO Evaluation

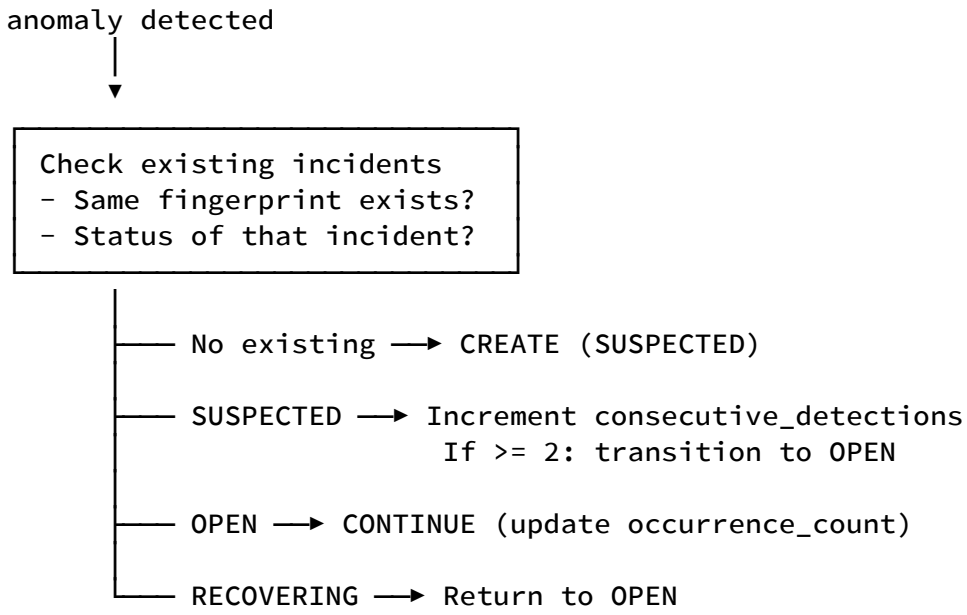
Each anomaly is evaluated against SLO thresholds:

```
For each detected anomaly:
    1. Evaluate latency vs SLO
    2. Evaluate error rate vs SLO
    3. Evaluate database latency vs baseline
    4. Check request rate (surge/cliff)
    5. Determine combined SLO status
    6. Adjust severity if needed
```

See SLO Evaluation for details.

Phase 6: Incident Lifecycle

Anomalies enter the incident lifecycle:



See Incident Lifecycle for details.

Output Structure

Final output for each service:

```
{
  "alert_type": "anomaly_detected",
  "service_name": "booking",
  "timestamp": "2024-01-15T10:30:00",
  "time_period": "business_hours",
  "overall_severity": "high",
  "anomaly_count": 1,

  "anomalies": {
    "latency_spike_recent": {
      "type": "consolidated",
      "severity": "high",
      "detection_signals": [...],
      "cascade_analysis": {...}
    }
  },

  "current_metrics": {...},
  "slo_evaluation": {...},
  "fingerprinting": {...}
}
```

Error Handling

Metrics Unavailable

When VictoriaMetrics is unreachable:

```
{
  "alert_type": "metrics_unavailable",
  "service_name": "booking",
  "error": "Metrics collection failed",
  "failed_metrics": ["request_rate", "application_latency"],
  "skipped_reason": "critical_metrics_unavailable"
}
```

Detection is skipped to prevent false alerts from missing data.

Model Not Found

When no trained model exists:

```
{
  "alert_type": "error",
  "service_name": "new-service",
  "error_message": "No trained model found for time period"
}
```

Run training to create models for new services.

SLO Evaluation Layer

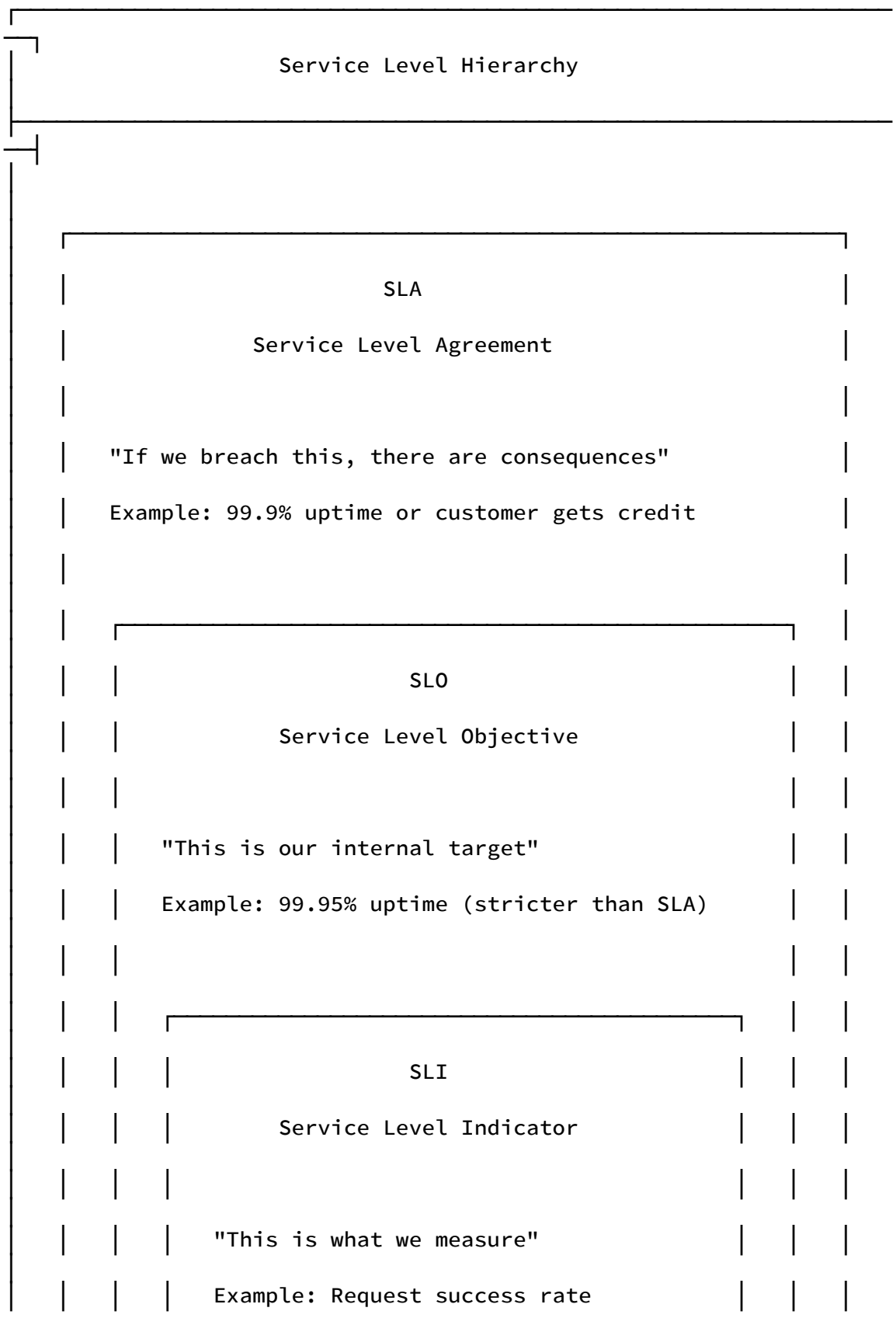
The SLO (Service Level Objective) evaluation layer adjusts anomaly severity based on operational thresholds. This chapter explains what SLOs are, why they matter for anomaly detection, and how Yaga2 uses them.

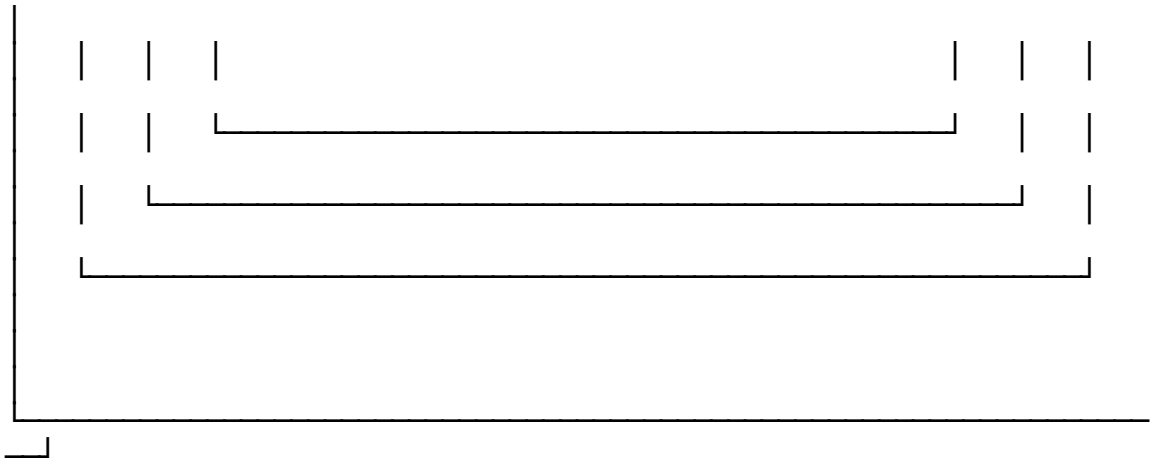
What is an SLO?

SLO (Service Level Objective) is a target level of service reliability that you promise to maintain. It's the answer to: *"What does 'good enough' look like for this service?"*

The SLI → SLO → SLA Hierarchy

Understanding SLOs requires understanding the full hierarchy:


Service Level Hierarchy
SLA
Service Level Agreement
"If we breach this, there are consequences"
Example: 99.9% uptime or customer gets credit
SLO
Service Level Objective
"This is our internal target"
Example: 99.95% uptime (stricter than SLA)
SLI
Service Level Indicator
"This is what we measure"
Example: Request success rate



Term	Definition	Example
SLI (Indicator)	The metric you measure	Request latency in milliseconds
SLO (Objective)	The target value for that metric	99% of requests < 500ms
SLA (Agreement)	Legal/business commitment with consequences	99.9% uptime or refund

Common SLO Types

SLO Type	What It Measures	Example Target
Availability	Is the service responding?	99.9% of requests succeed
Latency	How fast does it respond?	95% of requests < 300ms
Error Rate	How many requests fail?	< 0.1% error rate
Throughput	How much can it handle?	1000 req/s capacity

Why SLOs Matter

Without SLOs, you have two problems:

Problem 1: Over-alerting

```
❌ Static threshold: Alert if latency > 200ms
   ↓
   You get 50 alerts per day for "normal" spikes
   ↓
   Alert fatigue → Team ignores alerts
   ↓
   Real incidents get missed
```

Problem 2: Under-alerting

```
❌ ML-only detection: Alert when "unusual"
   ↓
   Latency at 400ms is unusual (normally 100ms)
   ↓
   But users don't notice 400ms latency
   ↓
   Alert noise for non-issues
```

The SLO Solution:

```
✅ SLO-aware detection:
   ↓
   ML detects 400ms latency as unusual
   ↓
   SLO evaluation: 400ms < 500ms acceptable threshold
   ↓
   Severity adjusted to "low" (no page, just log)
   ↓
   Team focuses on real problems
```

Why SLOs Matter for Anomaly Detection

ML-based anomaly detection answers: **“Is this unusual?”**

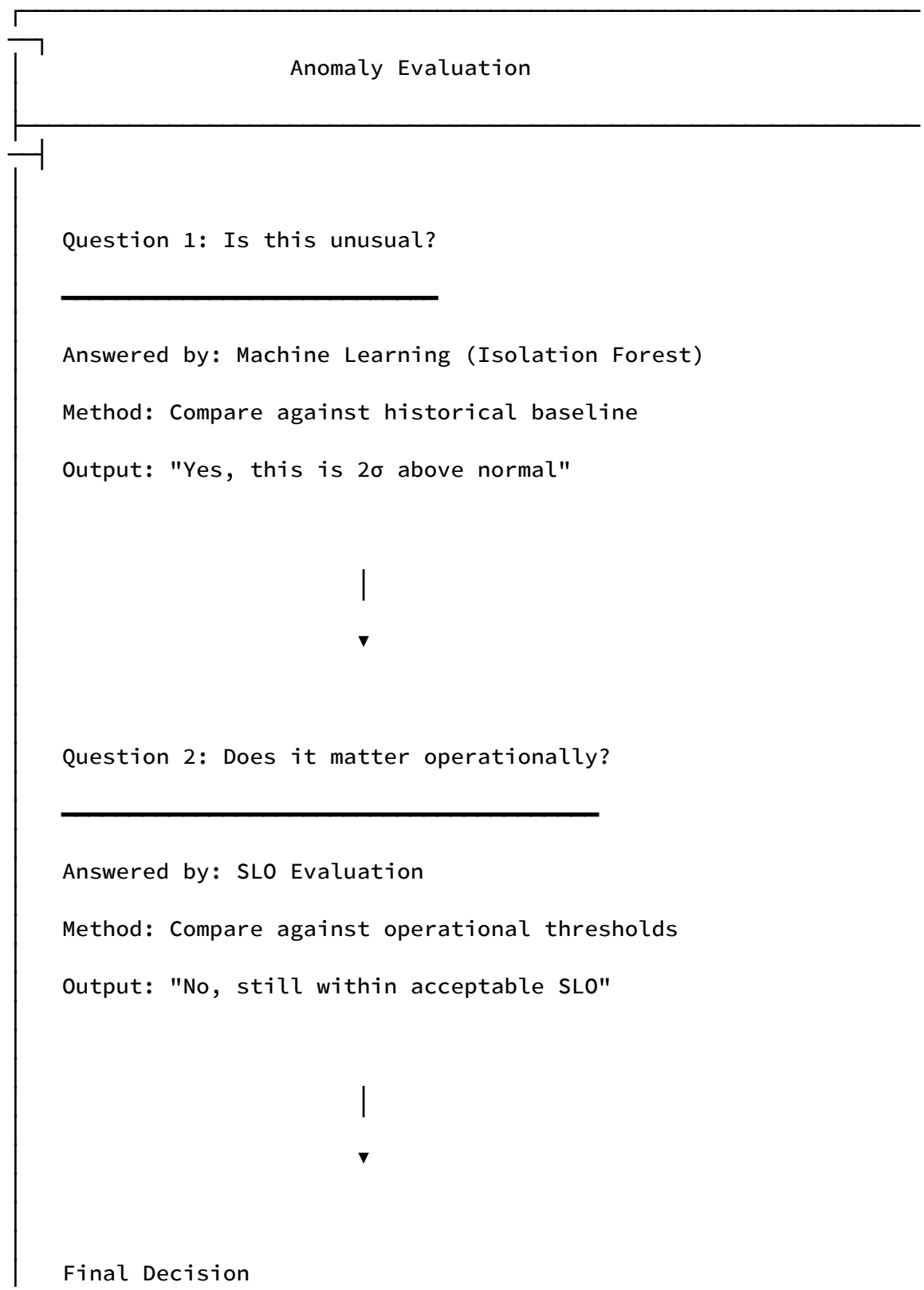
But unusual doesn’t mean bad:

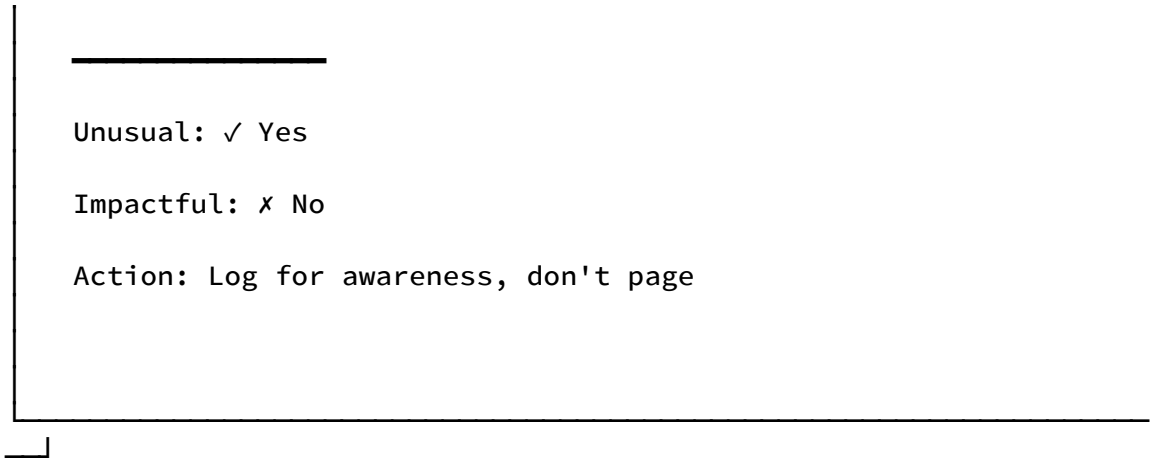
- A traffic spike during a sale is unusual but good

- Latency at 300ms when it's usually 100ms is unusual but might be acceptable
- Error rate dropping to 0% is unusual but definitely not a problem

SLO evaluation answers a different question: **"Does this impact users?"**

The Two-Question Model





Example: The 280ms Latency Case

Consider a booking service:

Metric	Value
Current latency	280ms
Historical baseline	150ms
Statistical deviation	+2.3σ (p91 percentile)

ML-only approach:

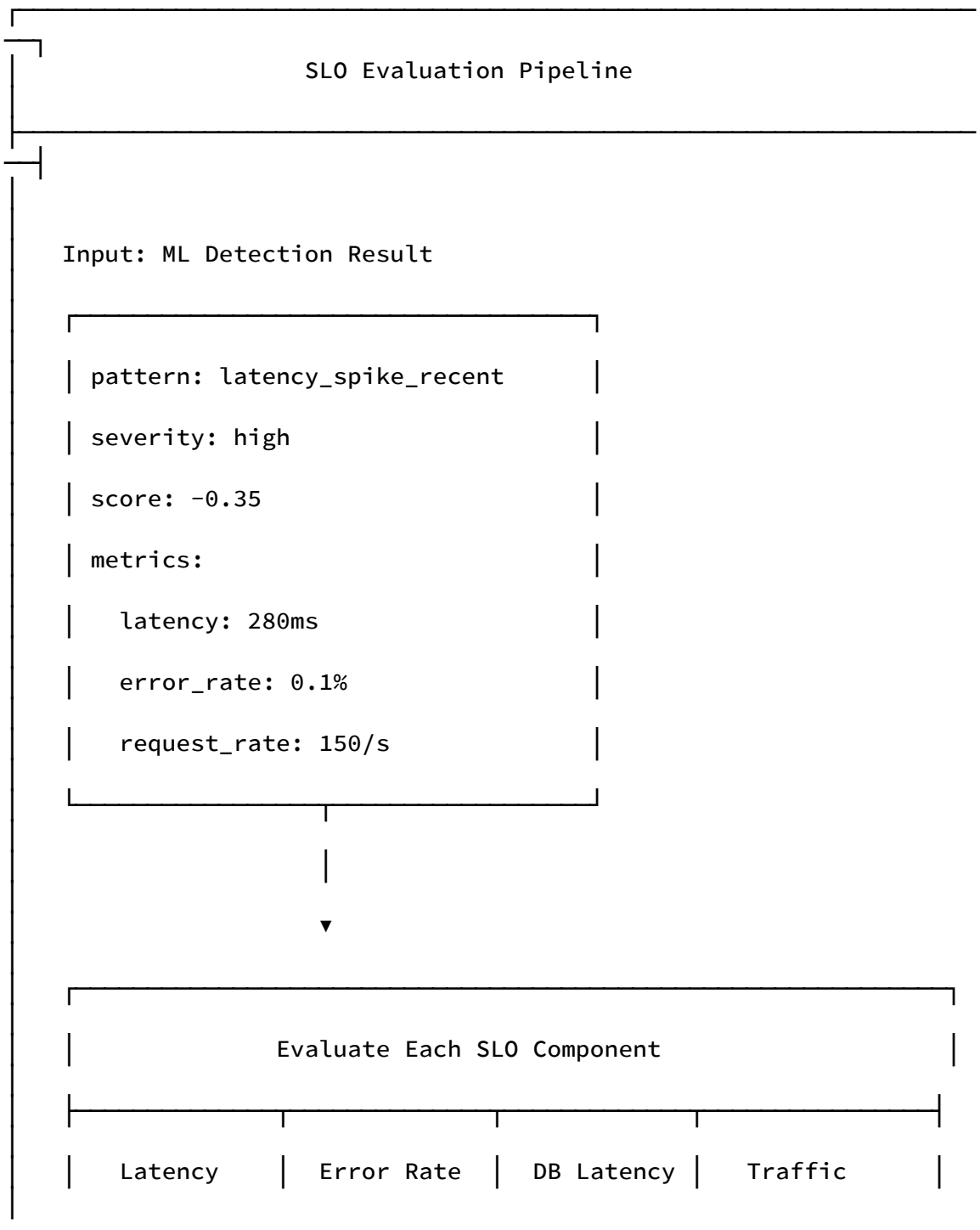
```
ML says: "280ms is unusual! That's 2.3σ above normal!"
Result: HIGH severity alert
Problem: Engineer pages at 3 AM for non-issue
```

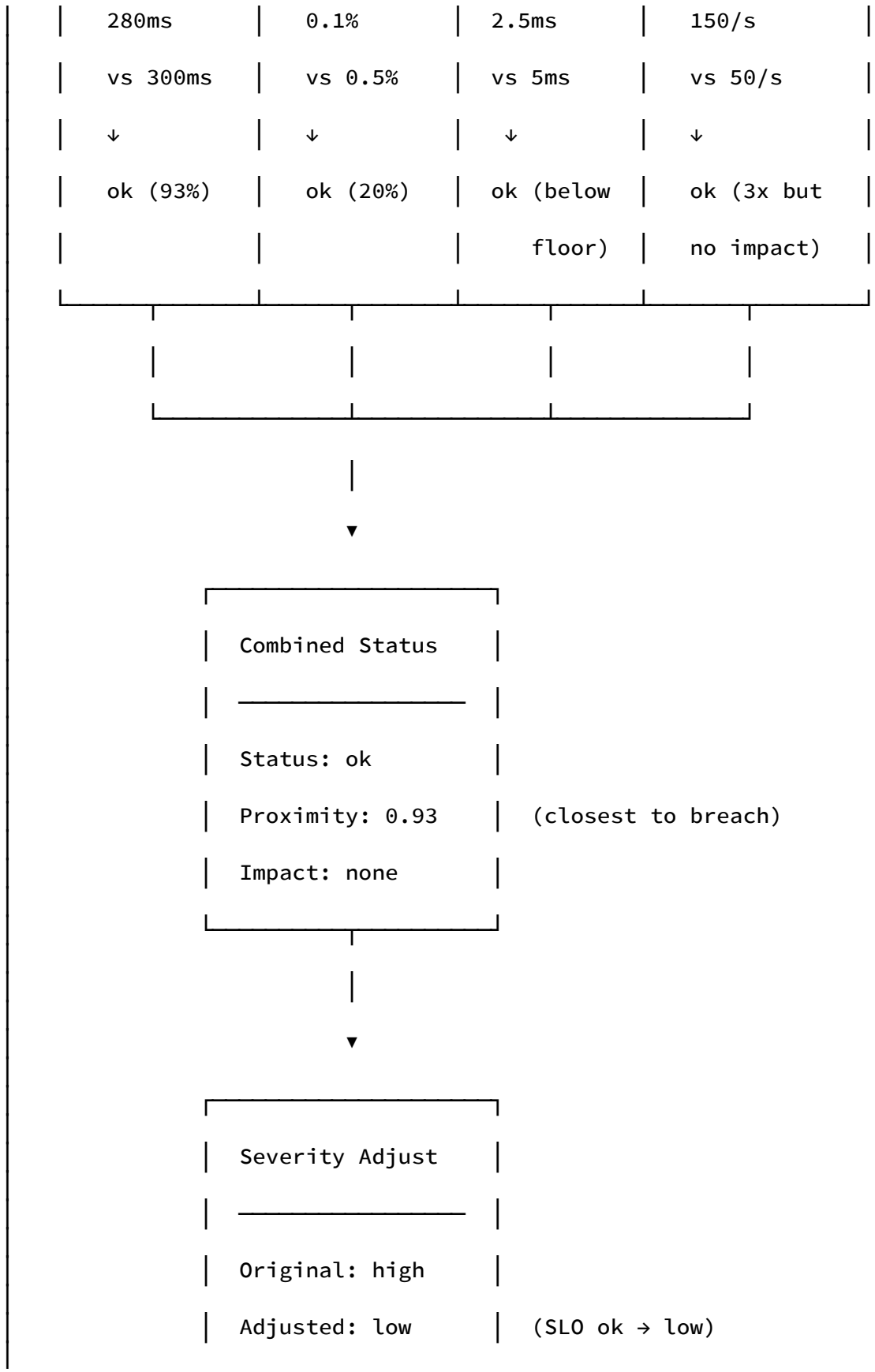
SLO-aware approach:

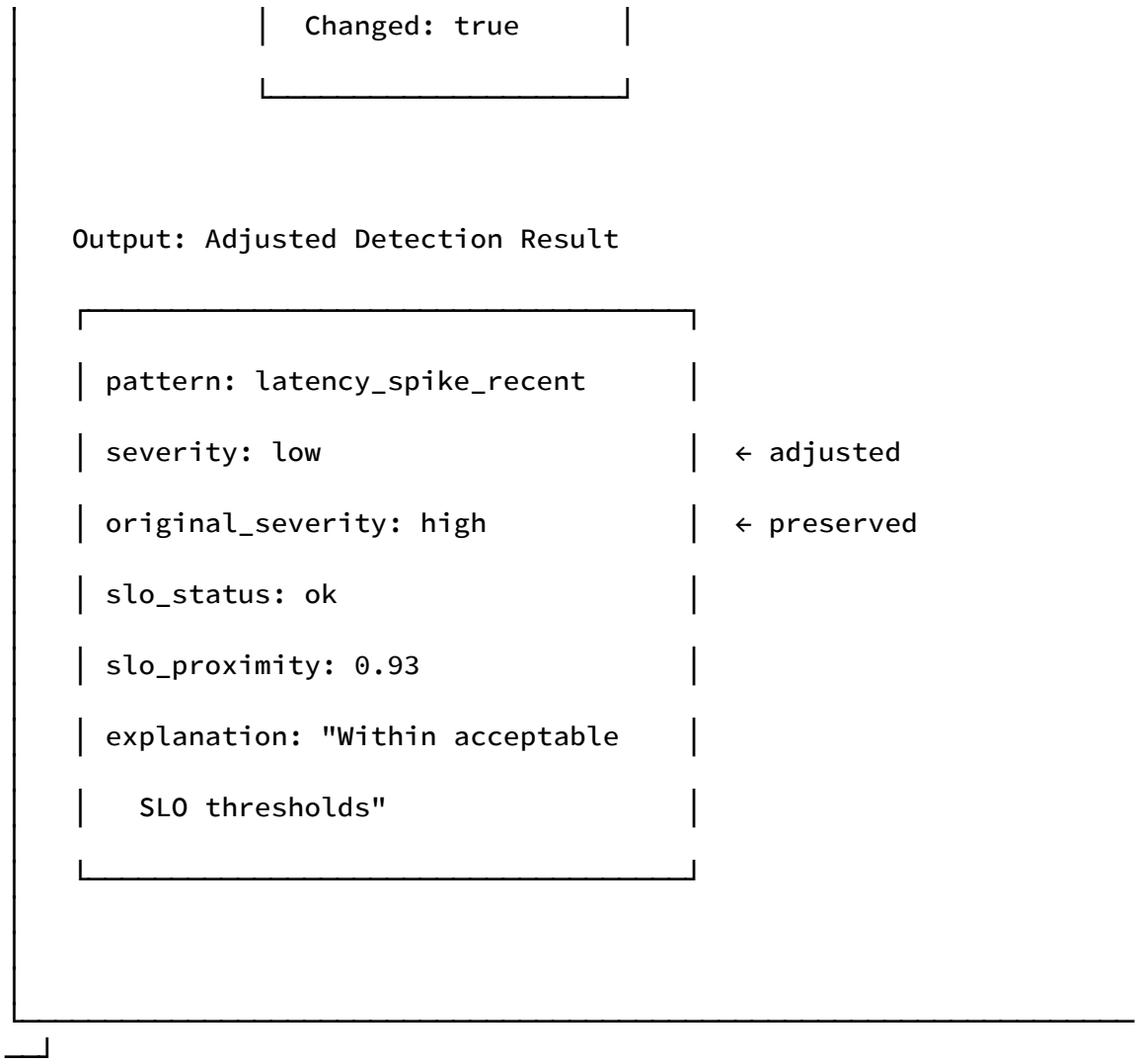
```
ML says: "280ms is unusual"
SLO says: "280ms < 300ms acceptable threshold"
SLO says: "280ms is 93% toward the limit (proximity = 0.93)"
Result: LOW severity (logged but no page)
Benefit: Engineer sleeps, system still tracks the deviation
```

How Yaga2's SLO Layer Works

Evaluation Flow




280ms
vs 300ms
↓
ok (93%)
0.1%
vs 0.5%
↓
ok (20%)
2.5ms
vs 5ms
↓
ok (below floor)
150/s
vs 50/s
↓
ok (3x but no impact)
Combined Status
Status: ok
Proximity: 0.93
Impact: none
(closest to breach)
Severity Adjust
Original: high
Adjusted: low
(SLO ok → low)



SLO Components in Yaga2

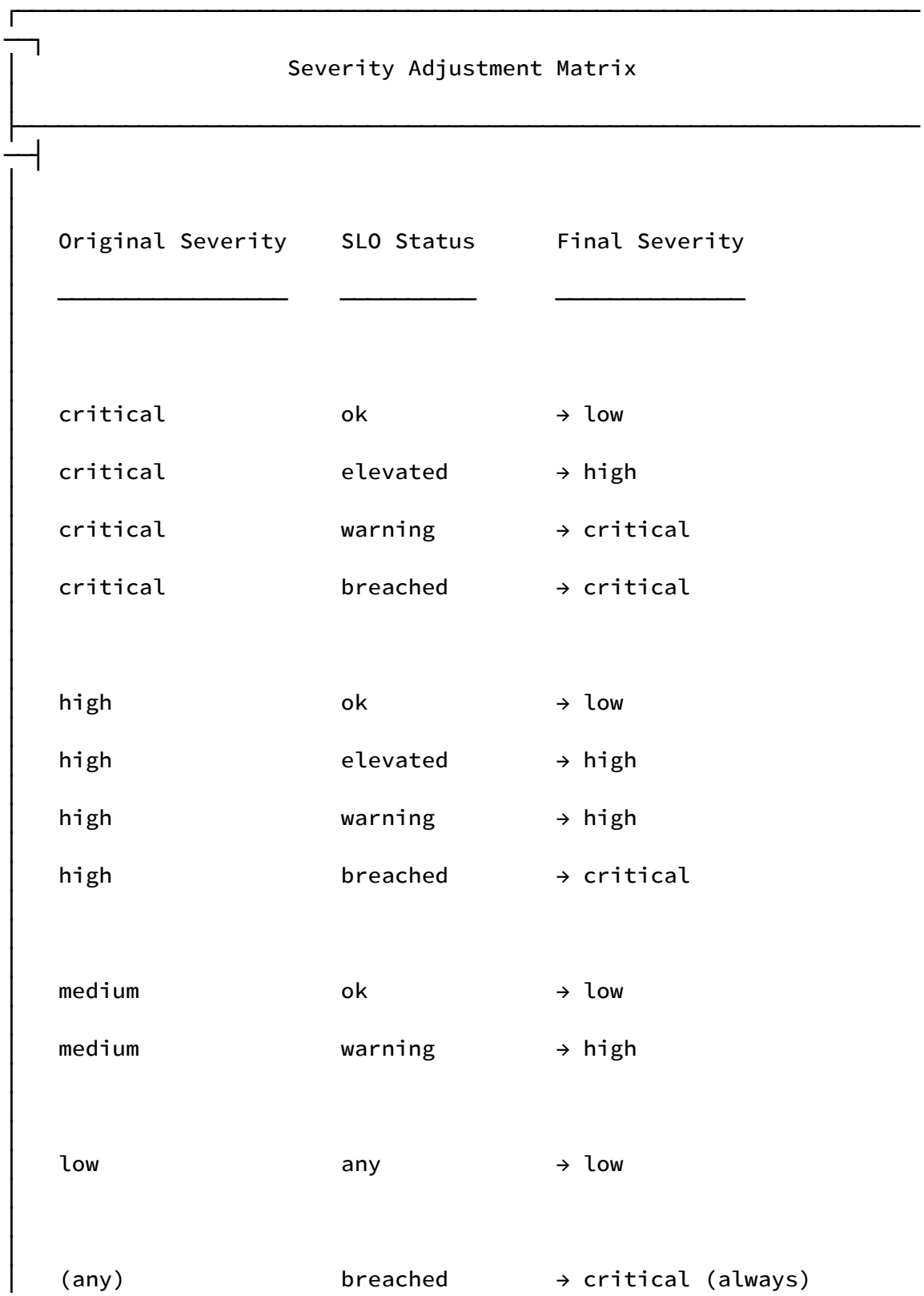
Component	Threshold Type	Evaluation Method
Latency	Absolute (ms)	Compare against acceptable/warning/critical thresholds
Error Rate	Absolute (%)	Compare against acceptable/warning/critical with floor suppression

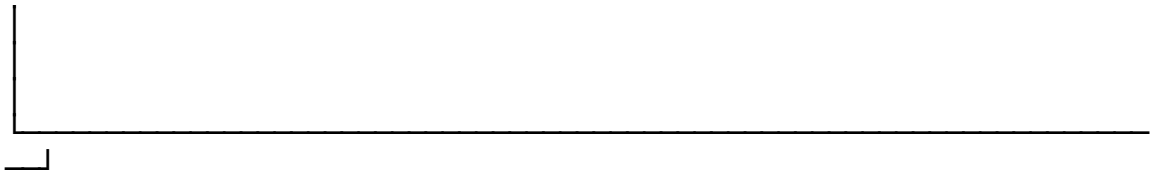
Component	Threshold Type	Evaluation Method
Database Latency	Ratio-based	Compare ratio against training baseline (1.5x, 2x, 3x, 5x)
Request Rate	Ratio-based	Detect surge (≥200%) or cliff (≤50%) with correlation

SLO Status

Status	What It Means	Severity Impact
`ok`	All metrics within acceptable limits	Severity → `low`
`elevated`	Above acceptable, below warning	Severity stays as-is
`warning`	Approaching SLO breach	Severity → `high`
`breached`	SLO threshold exceeded	Severity → `critical`

Severity Adjustment Rules





Key Insight: When SLO status is `ok`, severity is **always** adjusted to `low`, regardless of the original ML-assigned severity. This ensures alerts reflect operational impact, not just statistical deviation.

Practical Examples

Example 1: Acceptable Anomaly

```
Service: booking
─────────────────────────────────────────────────────────────

Current Metrics:
  latency: 420ms (usually 200ms)
  error_rate: 0.2%

ML Detection:
  Pattern: latency_spike_recent
  Severity: high
  Reason: Latency is 3σ above baseline

SLO Evaluation:
  Latency: 420ms < 500ms acceptable ✓
  Errors: 0.2% < 0.5% acceptable ✓
  Status: ok

Final Decision:
  Severity: low
  Action: Log for awareness, no page
  Message: "Anomaly detected but within SLO"
```

Example 2: Warning Zone

```
Service: search
──────────────────────────────────────────────────────────

Current Metrics:
  latency: 750ms (acceptable: 500ms, warning: 800ms)
  error_rate: 0.8% (acceptable: 0.5%, warning: 1%)

ML Detection:
  Pattern: latency_spike_recent
  Severity: high

SLO Evaluation:
  Latency: 750ms > 500ms acceptable, < 800ms warning
  Errors: 0.8% > 0.5% acceptable, < 1% warning
  Status: warning
  Proximity: 0.94 (close to warning threshold)

Final Decision:
  Severity: high (maintained)
  Action: Alert on-call for investigation
  Message: "Approaching SLO limits"
```

Example 3: SLO Breach

```
Service: checkout
──────────────────────────────────────────────────────────

Current Metrics:
  latency: 2500ms (critical: 1000ms)
  error_rate: 5%

ML Detection:
  Pattern: traffic_surge_failing
  Severity: critical

SLO Evaluation:
  Latency: 2500ms > 1000ms critical ✗
  Errors: 5% > 2% critical ✗
  Status: breached

Final Decision:
  Severity: critical (confirmed)
  Action: Immediate page, potential incident
  Message: "SLO breached - users impacted"
```

Configuration

Default SLO Thresholds

```
{
  "slos": {
    "enabled": true,
    "allow_downgrade_to_informational": true,
    "require_slo_breach_for_critical": true,
    "defaults": {
      "latency_acceptable_ms": 500,
      "latency_warning_ms": 800,
      "latency_critical_ms": 1000,
      "error_rate_acceptable": 0.005,
      "error_rate_warning": 0.01,
      "error_rate_critical": 0.02,
      "error_rate_floor": 0.001,
      "database_latency_floor_ms": 5.0,
      "database_latency_ratios": {
        "info": 1.5,
        "warning": 2.0,
        "high": 3.0,
        "critical": 5.0
      }
    }
  }
}
```

Per-Service Overrides

Different services have different requirements:

```
{
  "slos": {
    "services": {
      "checkout": {
        "latency_acceptable_ms": 300,
        "latency_critical_ms": 500,
        "error_rate_acceptable": 0.001
      },
      "search": {
        "latency_acceptable_ms": 200,
        "latency_critical_ms": 400
      },
      "admin-panel": {
        "latency_acceptable_ms": 2000,
        "error_rate_acceptable": 0.01
      }
    }
  }
}
```

Busy Period Configuration

During high-traffic periods (holidays, sales), thresholds can be automatically relaxed:

```
{
  "slos": {
    "defaults": {
      "busy_period_factor": 1.5
    },
    "busy_periods": [
      {
        "start": "2024-12-20T00:00:00",
        "end": "2025-01-05T23:59:59"
      }
    ]
  }
}
```

During busy periods:

- 500ms acceptable → 750ms acceptable

- 0.5% error acceptable → 0.75% error acceptable

Best Practices

1. Set SLOs Based on User Experience

```
❌ Bad: "Let's set latency SLO at 100ms because it sounds good"
   Problem: Your P50 latency is already 150ms

✓ Good: "Users report issues above 500ms, let's set acceptable at
400ms"
   Benefit: SLO reflects actual user impact
```

2. Use the Error Budget Concept

If your SLO is 99.9% availability:

- Error budget = 0.1% of requests can fail
- Monthly budget ≈ 43 minutes of downtime
- Use this to decide when to deploy vs stabilize

3. Have Different SLOs for Different Services

Service	Latency SLO	Why
Checkout	300ms	Revenue-critical, users waiting
Search	200ms	User experience, interactive
Admin Panel	2000ms	Internal tool, less critical
Batch Jobs	30000ms	Background, no users waiting

4. Review and Adjust SLOs Quarterly

- Analyze false positive rate
- Check if SLOs match user complaints
- Tighten SLOs as systems improve

Further Reading

- Latency Evaluation - Response time SLO checking
- Error Rate Evaluation - Error percentage SLO checking
- Database Latency - Ratio-based DB latency evaluation
- Request Rate - Traffic surge and cliff detection
- Severity Adjustment - How severity is adjusted based on SLO status

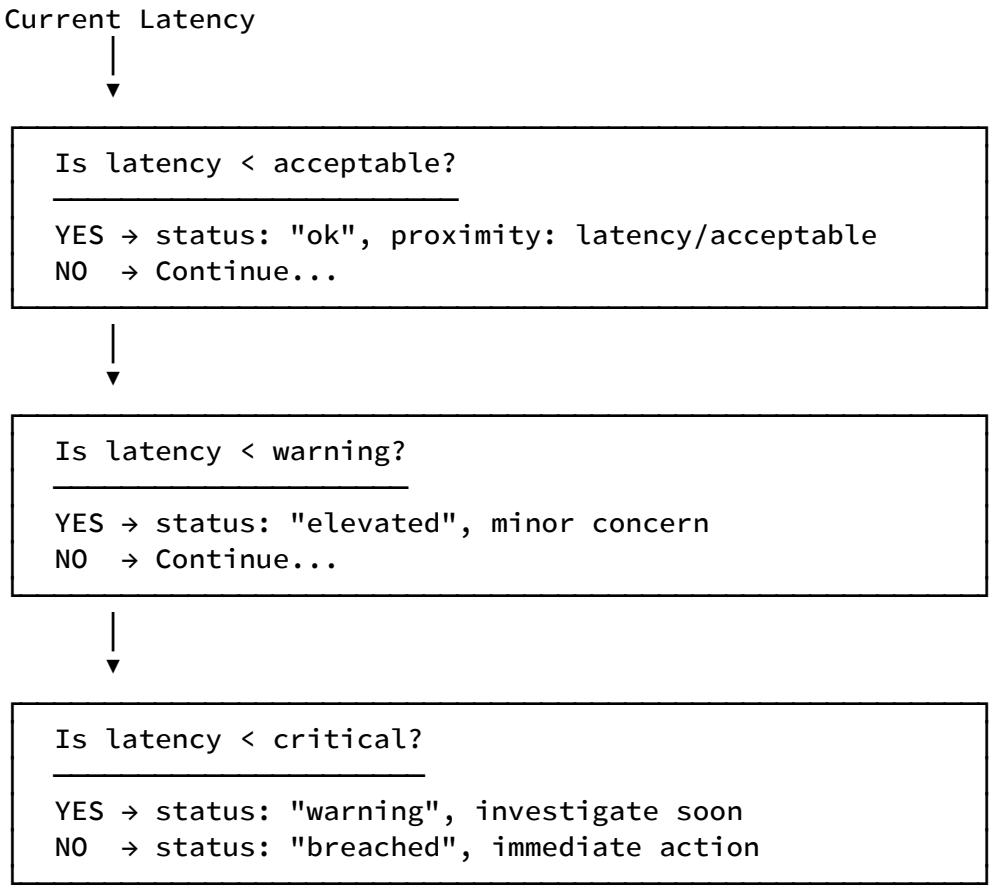
Latency Evaluation

Latency evaluation compares current response time against SLO thresholds.

Thresholds

Level	Default	Meaning
`acceptable`	500ms	Latency below this is fine
`warning`	800ms	Approaching SLO limit
`critical`	1000ms	SLO breach

Evaluation Logic



Proximity Score

The `proximity` value indicates how close to breach (0.0 - 1.0+):

Proximity	Interpretation
0.0 - 0.5	Comfortable margin
0.5 - 0.8	Getting close
0.8 - 1.0	Near threshold
> 1.0	Threshold exceeded

Output Example

```
{
  "latency_evaluation": {
    "status": "warning",
    "value": 750.0,
    "threshold_acceptable": 500,
    "threshold_warning": 800,
    "threshold_critical": 1000,
    "proximity": 0.94
  }
}
```

Per-Service Configuration

Critical services can have stricter thresholds:

```
{
  "slos": {
    "services": {
      "booking": {
        "latency_acceptable_ms": 300,
        "latency_warning_ms": 400,
        "latency_critical_ms": 500
      },
      "search": {
        "latency_acceptable_ms": 200,
        "latency_critical_ms": 400
      }
    }
  }
}
```

Busy Period Handling

During configured busy periods, thresholds are relaxed by `busy_period_factor`:

```
{
  "slos": {
    "defaults": {
      "busy_period_factor": 1.5
    },
    "busy_periods": [
      {"start": "2024-12-20T00:00:00", "end": "2025-01-05T23:59:59"}
    ]
  }
}
```

During busy periods:

- 500ms acceptable → 750ms acceptable
- 1000ms critical → 1500ms critical

Error Rate Evaluation

Error rate evaluation compares current error percentage against SLO thresholds.

Thresholds

Level	Default	Meaning
`acceptable`	0.5%	Error rate below this is fine
`warning`	1.0%	Approaching SLO limit
`critical`	2.0%	SLO breach

Error Rate Floor (Suppression)

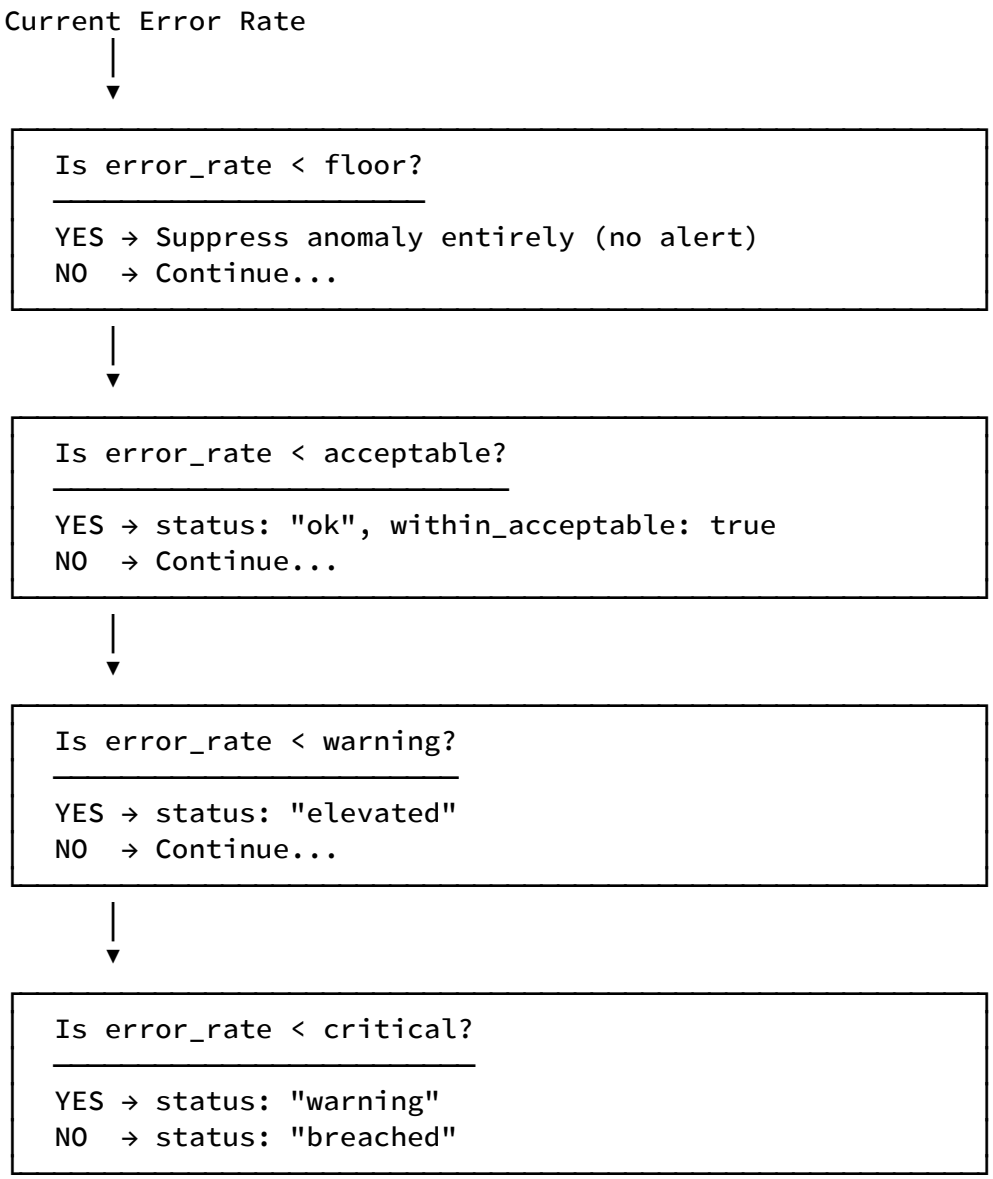
To prevent alert noise from tiny error rate deviations, an optional `error_rate_floor` can suppress anomalies when errors are operationally insignificant.

```
{
  "slos": {
    "services": {
      "booking": {
        "error_rate_acceptable": 0.002,
        "error_rate_floor": 0.002
      }
    }
  }
}
```

Error Rate	Floor (0.2%)	Result
0.01%	Below floor	**Suppressed** (no alert)

Error Rate	Floor (0.2%)	Result
0.15%	Below floor	**Suppressed** (no alert)
0.25%	Above floor	Alert fires

Evaluation Logic



Output Example


```
{
  "error_rate_evaluation": {
    "status": "ok",
    "value": 0.001,
    "value_percent": "0.10%",
    "threshold_acceptable": 0.005,
    "threshold_warning": 0.01,
    "threshold_critical": 0.02,
    "within_acceptable": true
  }
}
```


When Floor is Active


```
{
  "slo_context": {
    "current_value": 0.005,
    "current_value_percent": "0.50%",
    "acceptable_threshold": 0.002,
    "critical_threshold": 0.01,
    "suppression_threshold": 0.002,
    "within_acceptable": false
  }
}
```

Per-Service Configuration

```
{
  "slos": {
    "services": {
      "booking": {
        "error_rate_acceptable": 0.002,
        "error_rate_warning": 0.005,
        "error_rate_critical": 0.01,
        "error_rate_floor": 0.002
      },
      "admin-api": {
        "error_rate_acceptable": 0.01,
        "error_rate_critical": 0.05
      }
    }
  }
}
```

Exception Enrichment

When error SLO is breached (HIGH or CRITICAL severity), exception context is automatically queried and added to the alert:

```
{
  "exception_context": {
    "service_name": "search",
    "total_exception_rate": 0.35,
    "top_exceptions": [
      {"type": "R2D2Exception", "rate": 0.217, "percentage": 62.0},
      {"type": "UserInputException", "rate": 0.083, "percentage":
23.7}
    ]
  }
}
```

This helps identify which exception types are causing the error spike.

Database Latency Evaluation

Database latency uses a **hybrid approach**: noise floor filtering + ratio-based thresholds against training baseline.

Why Ratio-Based?

Unlike application latency with fixed SLO targets, database latency varies significantly:

- A fast service might have 5ms DB latency normally
- A reporting service might have 500ms DB latency normally

Ratio-based thresholds adapt to each service's baseline.

Noise Floor

Very low database latency values are filtered as operationally meaningless:

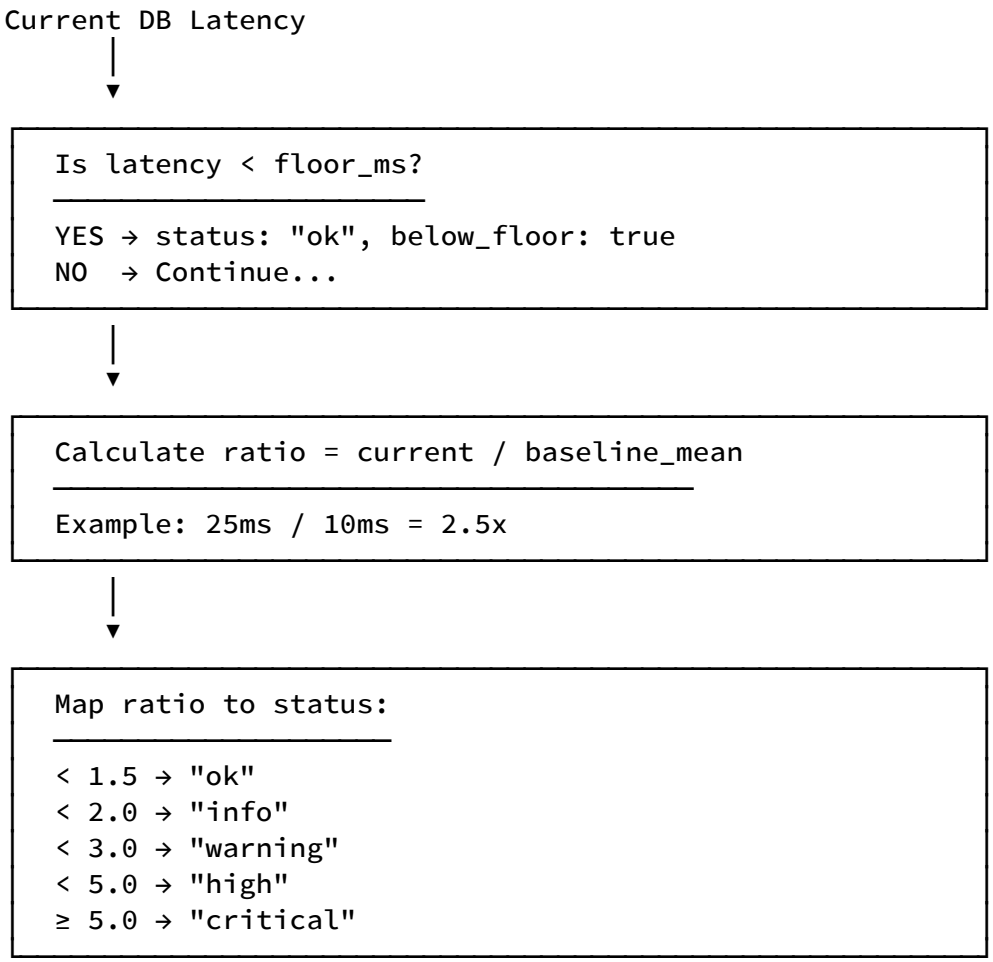
Latency	Floor (5ms)	Result
2ms	Below floor	Always `ok`
0.3ms → 0.4ms	Below floor	Ignored
10ms	Above floor	Evaluate ratio

This prevents alerts for sub-millisecond changes that the ML might flag as anomalous.

Ratio Thresholds

Status	Ratio	Meaning
`ok`	< 1.5x	Within normal variance
`info`	1.5x - 2x	Slightly elevated
`warning`	2x - 3x	Elevated, investigate
`high`	3x - 5x	Significantly elevated
`critical`	≥ 5x	SLO breach

Evaluation Logic



Output Examples

Below Floor


```
{
  "database_latency_evaluation": {
    "status": "ok",
    "value_ms": 2.0,
    "baseline_mean_ms": 1.0,
    "ratio": 0.0,
    "below_floor": true,
    "floor_ms": 5.0,
    "explanation": "Below noise floor (2.0ms < 5.0ms)"
  }
}
```


Elevated Ratio


```
{
  "database_latency_evaluation": {
    "status": "warning",
    "value_ms": 25.0,
    "baseline_mean_ms": 10.0,
    "ratio": 2.5,
    "below_floor": false,
    "floor_ms": 5.0,
    "thresholds": {
      "info": 1.5,
      "warning": 2.0,
      "high": 3.0,
      "critical": 5.0
    },
    "explanation": "DB latency elevated: 25.0ms is 2.5x baseline
(10.0ms)"
  }
}
```

Per-Service Configuration

Services with different DB performance characteristics can have custom thresholds:

```
{
  "slos": {
    "services": {
      "search": {
        "database_latency_floor_ms": 2.0,
        "database_latency_ratios": {
          "info": 1.3,
          "warning": 1.5,
          "high": 2.0,
          "critical": 3.0
        }
      },
      "reporting": {
        "database_latency_floor_ms": 50.0,
        "database_latency_ratios": {
          "info": 2.0,
          "warning": 3.0,
          "high": 5.0,
          "critical": 10.0
        }
      }
    }
  }
}
```

Relationship to Pattern Matching

Database latency evaluation complements pattern matching:

Pattern	Database Latency Evaluation
`database_degradation`	DB ratio 2-3x, compensating
`database_bottleneck`	DB ratio ≥3x, dominant latency

The SLO evaluation provides the **ratio context** that pattern matching uses for severity.

Request Rate Evaluation (Surge/Cliff)

Request rate evaluation detects significant traffic changes: **surges** (spikes) and **cliffs** (drops).

Key Insight

Traffic anomalies use **correlation-based severity**:

- A surge alone is often benign (marketing campaign, organic growth)
- A surge becomes problematic when causing SLO issues
- A cliff is inherently concerning (may indicate upstream failure)

Thresholds

Type	Threshold	Default
Surge	≥ 200% of baseline	2x normal traffic
Cliff	≤ 50% of baseline	Half normal traffic

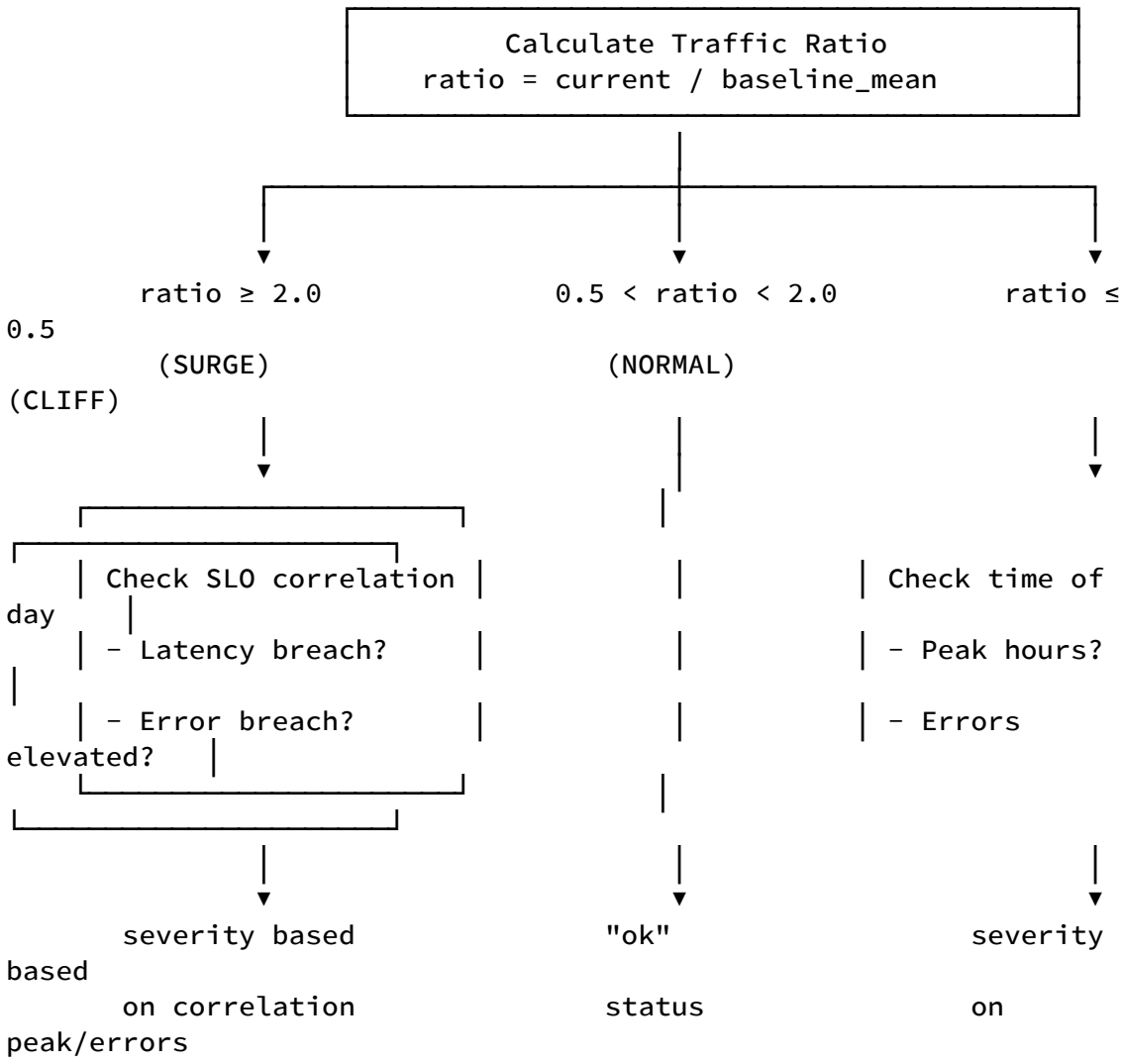
Surge Severity Logic

Condition	Severity	Rationale
Surge only	`informational`	Normal growth or campaign
Surge + latency SLO breach	`warning`	Capacity stress
Surge + error SLO breach	`high`	Active incident

Cliff Severity Logic

Condition	Severity	Rationale
Cliff off-peak	`warning`	May be expected
Cliff peak hours	`high`	Likely incident
Cliff + errors	`critical`	Confirmed incident

Evaluation Flow



Output Examples

Surge (Standalone)


```
{
  "request_rate_evaluation": {
    "status": "info",
    "type": "surge",
    "severity": "informational",
    "value_rps": 250.0,
    "baseline_mean_rps": 100.0,
    "ratio": 2.5,
    "threshold_percent": 200.0,
    "correlated_with_latency": false,
    "correlated_with_errors": false,
    "explanation": "Traffic surge (2.5x baseline) without SLO
impact. Normal growth or campaign traffic."
  }
}
```


Surge with Latency Impact


```
{
  "request_rate_evaluation": {
    "status": "warning",
    "type": "surge",
    "severity": "warning",
    "value_rps": 350.0,
    "baseline_mean_rps": 100.0,
    "ratio": 3.5,
    "correlated_with_latency": true,
    "correlated_with_errors": false,
    "explanation": "Traffic surge (3.5x baseline) correlating with
latency SLO breach - capacity issue."
  }
}
```

Cliff (Peak Hours)


```
{
  "request_rate_evaluation": {
    "status": "high",
    "type": "cliff",
    "severity": "high",
    "value_rps": 30.0,
    "baseline_mean_rps": 100.0,
    "ratio": 0.3,
    "threshold_percent": 50.0,
    "is_peak_hours": true,
    "correlated_with_errors": false,
    "explanation": "Traffic cliff (0.3x baseline) during peak hours
- investigate potential incident."
  }
}
```


Cliff with Errors


```
{
  "request_rate_evaluation": {
    "status": "critical",
    "type": "cliff",
    "severity": "critical",
    "value_rps": 15.0,
    "baseline_mean_rps": 100.0,
    "ratio": 0.15,
    "is_peak_hours": true,
    "correlated_with_errors": true,
    "explanation": "Traffic cliff (0.15x baseline) with errors -
likely upstream failure or routing issue."
  }
}
```

Configuration

```
{
  "slos": {
    "defaults": {
      "request_rate_surge_threshold": 2.0,
      "request_rate_cliff_threshold": 0.5
    },
    "services": {
      "booking": {
        "request_rate_evaluation": {
          "surge": {
            "threshold": 3.0
          },
          "cliff": {
            "standalone_severity": "high",
            "peak_hours_severity": "critical"
          }
        }
      }
    }
  }
}
```

Relationship to Pattern Matching

Pattern	Request Rate Evaluation
`traffic_surge_healthy`	Surge, no SLO correlation
`traffic_surge_degrading`	Surge + latency correlation
`traffic_surge_failing`	Surge + error correlation
`traffic_cliff`	Cliff detected

Severity Adjustment

The final step of SLO evaluation: adjusting ML-assigned severity based on operational impact.

Core Principle

SLO status determines final severity, not ML confidence.

SLO Status	Final Severity	Rationale
`ok`	`low`	Anomaly detected but operationally acceptable
`warning`	`high`	Approaching limits, should investigate
`breached`	`critical`	SLO exceeded, requires action

Adjustment Matrix

ML Severity	SLO Status	Final Severity	Example
critical	ok	**low**	280ms latency (unusual but < 300ms SLO)
critical	warning	high	750ms latency (approaching 800ms SLO)
critical	breached	critical	1200ms latency (> 1000ms SLO)
high	ok	**low**	Same principle
high	warning	high	No change needed
high	breached	critical	Escalated

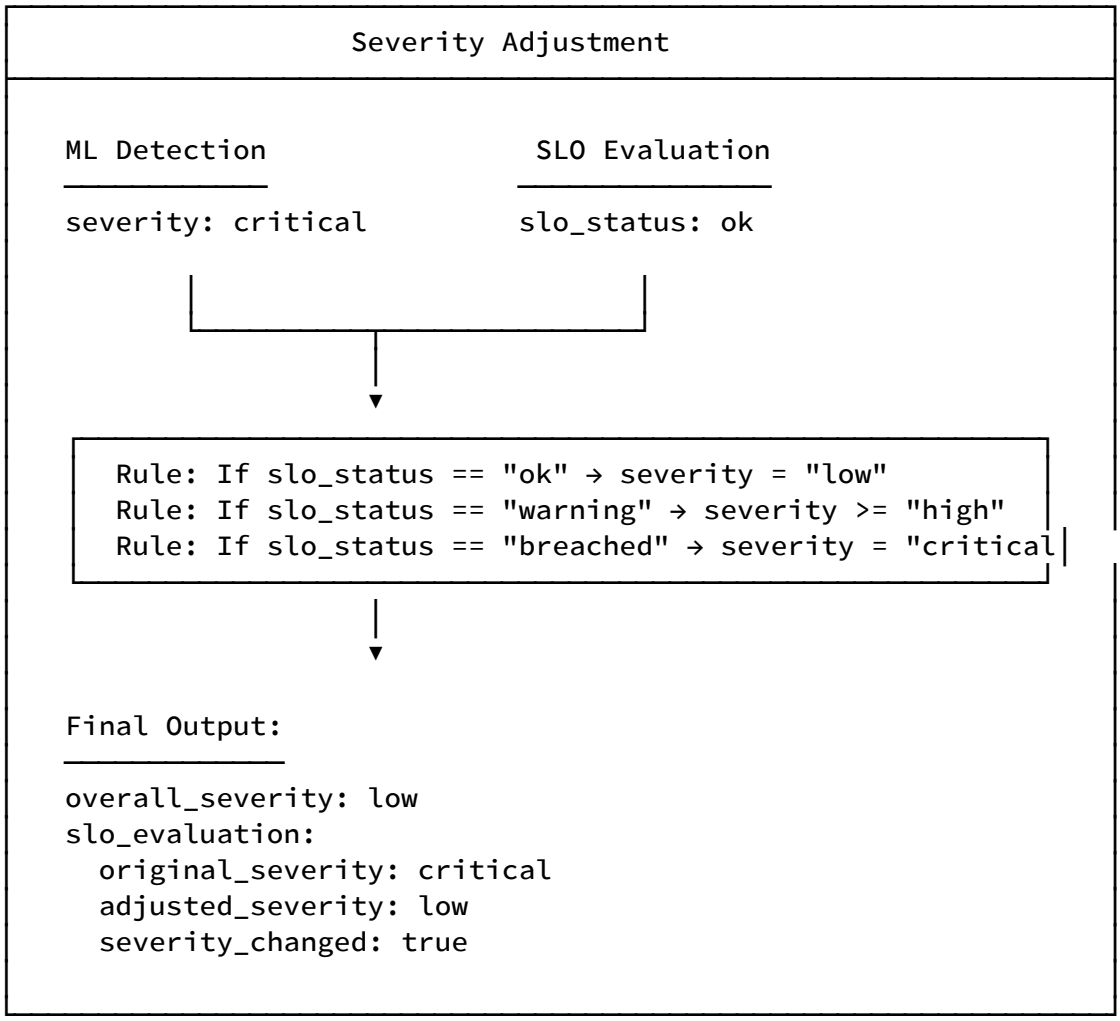
ML Severity	SLO Status	Final Severity	Example
medium	ok	**low**	Minor anomaly, within SLO
medium	warning	high	Escalated due to SLO proximity
low	breached	critical	Even low anomaly escalated if SLO breached

Key Behavior Change (v1.3.1)

Before v1.3.1: SLO `ok` adjusted to `medium` **After v1.3.1**: SLO `ok` adjusts to `low`

This ensures “operationally acceptable” consistently means “low priority.”

Adjustment Flow



Output Example

```
{
  "overall_severity": "low",
  "slo_evaluation": {
    "original_severity": "critical",
    "adjusted_severity": "low",
    "severity_changed": true,
    "slo_status": "ok",
    "slo_proximity": 0.56,
    "operational_impact": "informational",
    "explanation": "Severity adjusted from critical to low based on
SLO evaluation. Anomaly detected but metrics within acceptable SLO
thresholds (latency: 280ms < 300ms, errors: 0.10% < 0.50%)."
  }
}
```

Operational Impact Levels

Level	Meaning	Action
`none`	No anomaly or fully normal	No action
`informational`	Anomaly noted but acceptable	Log only
`actionable`	Approaching limits	Investigate
`critical`	SLO breached	Immediate action

Configuration Options

```
{
  "slos": {
    "enabled": true,
    "allow_downgrade_to_informational": true,
    "require_slo_breach_for_critical": true
  }
}
```

Option	Default	Effect
`enabled`	true	Enable/disable SLO evaluation
`allow_downgrade_to_informational`	true	Allow high/critical → low when SLO ok
`require_slo_breach_for_critical`	true	Only allow critical if SLO breached

root `overall_severity`

Important (v1.3.1): The root-level `overall_severity` field now correctly reflects the SLO-adjusted value.

```
{
  "overall_severity": "low",          // ← SLO-adjusted value
  "slo_evaluation": {
    "original_severity": "critical",  // ← ML-assigned value
    "adjusted_severity": "low"        // ← Same as overall_severity
  }
}
```

Previously, `overall_severity` could show `critical` while `adjusted_severity` showed `low`.

Alert Suppression

When SLO evaluation results in `low` severity:

- Alert is still generated (for logging/visibility)
- Alert may be filtered by downstream systems (e.g., skip PagerDuty)
- Dashboard shows alert with low priority indicator

To completely suppress alerts below a threshold, use downstream alert filtering rules.

Incident Lifecycle

The incident lifecycle manages anomaly state over time, reducing alert noise through confirmation requirements and grace periods. This chapter explains how Yaga2 tracks anomalies as they are detected, confirmed, and eventually resolved.

What Problem Does Incident Lifecycle Solve?

Raw anomaly detection produces a stream of “yes/no” decisions every inference cycle. Without state management, this creates significant operational problems:

Problem 1: Transient Spikes Cause Alert Noise

```
Without lifecycle management:
─────────────────────────────

10:00  Latency 450ms  → ALERT!  (ops team paged)
10:03  Latency 120ms  → Resolved
10:06  Latency 455ms  → ALERT!  (ops team paged again)
10:09  Latency 118ms  → Resolved
10:12  Latency 460ms  → ALERT!  (ops team paged again)
10:15  Latency 115ms  → Resolved

Result: 3 alerts in 15 minutes for what might be normal variance
        Ops team frustrated, starts ignoring alerts
```

Problem 2: Brief Recovery Causes Flapping

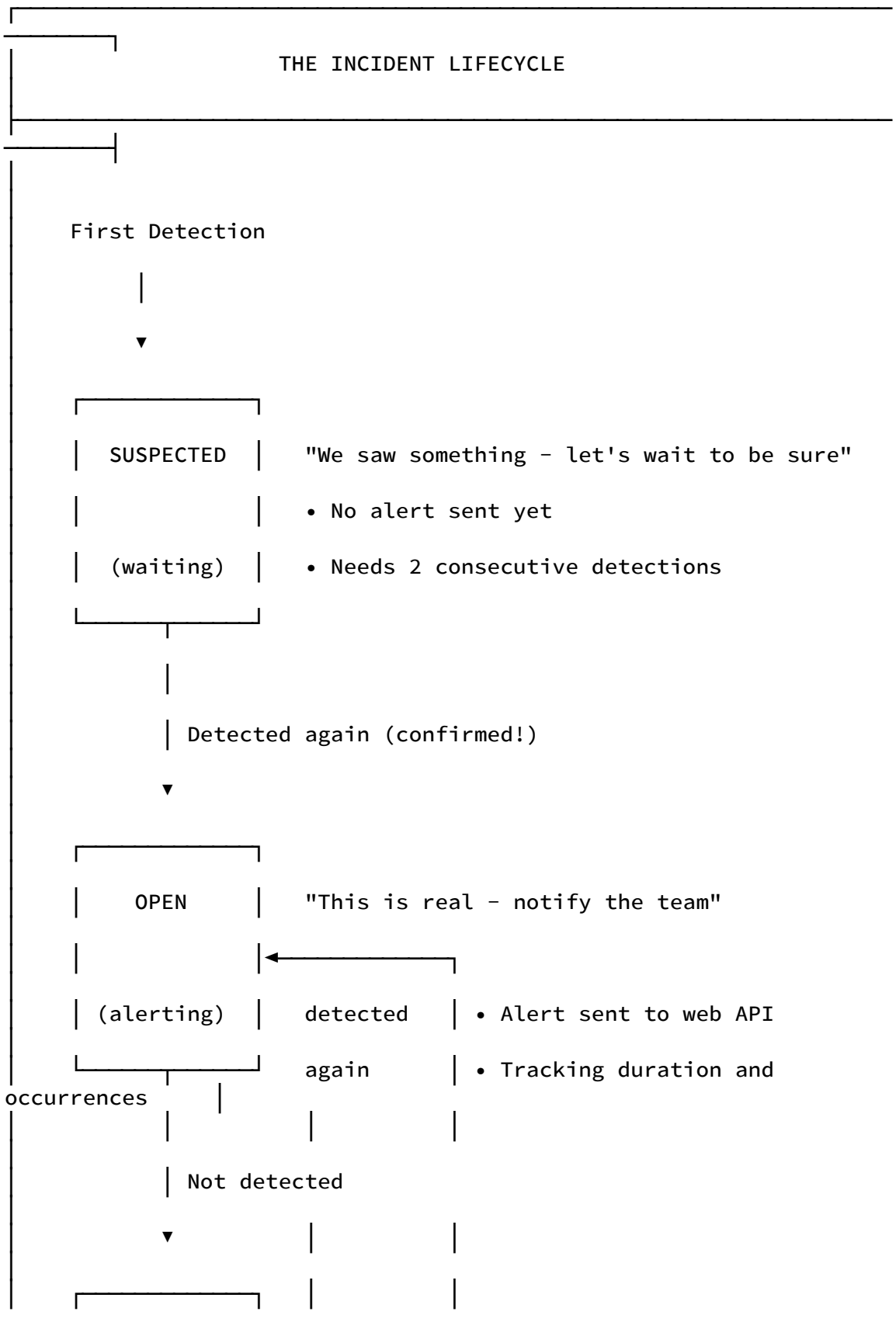
```
Without lifecycle management:
─────────────────────────────

10:00  Error rate 5%   → ALERT!
10:03  Error rate 5%   → Continue
10:06  Error rate 4.8% → Continue
10:09  Error rate 0.5% → Resolved  (brief dip)
10:12  Error rate 5.2% → ALERT!   (new alert!)
10:15  Error rate 5%   → Continue

Result: Same ongoing incident treated as two separate incidents
        Resolution was premature - the issue wasn't actually fixed
```

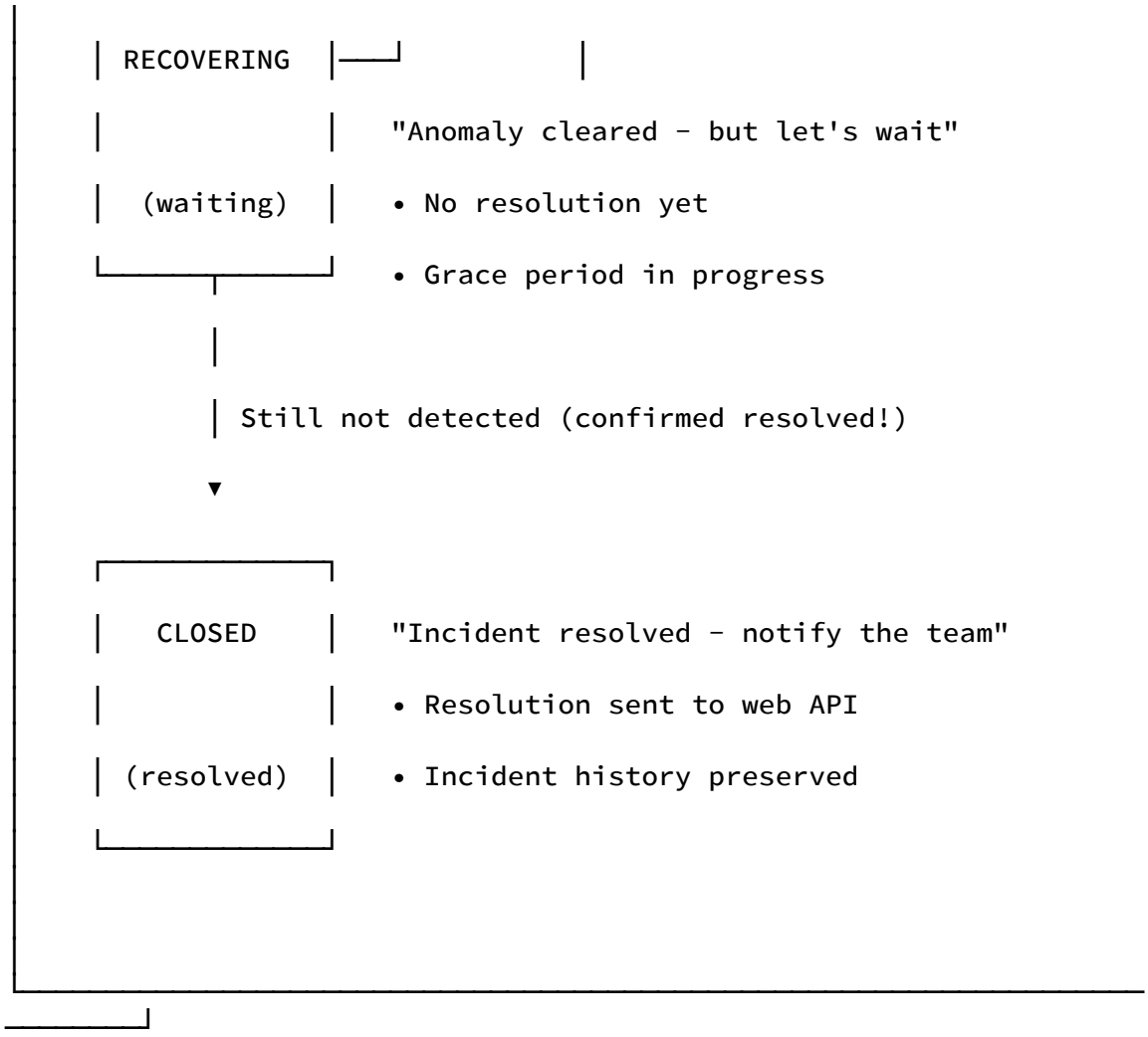
Problem 3: No Incident Correlation

```
Without lifecycle management:
─────────────────────────────

Each detection is independent. No way to answer:
- "How long has this been happening?"
- "Is this the same issue from yesterday?"
- "How many times has this pattern occurred?"
```

The Solution: Stateful Incident Tracking

Yaga2 solves these problems with a **state machine** that tracks anomalies through a defined lifecycle:


THE INCIDENT LIFECYCLE
First Detection
SUSPECTED
(waiting)
"We saw something - let's wait to be sure"
• No alert sent yet
• Needs 2 consecutive detections
Detected again (confirmed!)
OPEN
(alerting)
"This is real - notify the team"
detected
again
• Alert sent to web API
• Tracking duration and occurrences
Not detected



Core Concepts

Fingerprint vs Incident

These two concepts are central to incident tracking:

Concept	What It Is	Analogy
Fingerprint ID	Pattern identifier (same pattern = same ID)	The “type” of crime

Concept	What It Is	Analogy
Incident ID	Unique occurrence identifier	A specific “case file”

Fingerprint ID is **deterministic** - it’s computed from the anomaly content:

```
hash("booking_business_hours_latency_spike_recent")
  → anomaly_8d4a011b83ca
```

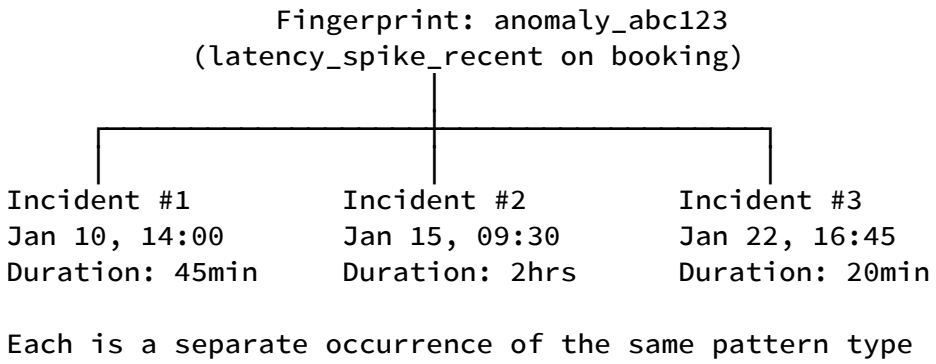
The same anomaly pattern on the same service always gets the same fingerprint ID.

Incident ID is **unique** - each new occurrence gets a fresh identifier:

```
Random UUID generation
  → incident_1dcbafc91480
```

Why Both IDs?

The same anomaly pattern can occur multiple times:



This enables powerful analysis:

- “This pattern has occurred 3 times this month”
- “Average duration is 58 minutes”
- “Longest incident was 2 hours on Jan 15”

States Explained

SUSPECTED - First Detection

When an anomaly is detected for the first time, it enters the SUSPECTED state:

STATE: SUSPECTED

What happened:

ML detected something unusual

What we do:

Wait for confirmation (not alert yet)

Why wait?

Single-point anomalies are often noise:

- Brief network blip
- One slow database query
- Measurement variance

Payload fields:

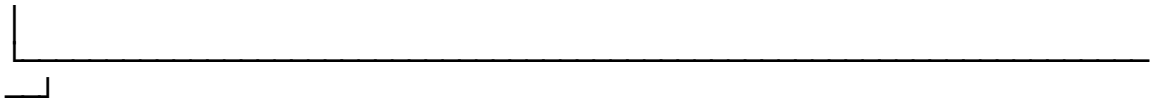
status: "SUSPECTED"

is_confirmed: false

confirmation_pending: true

cycles_to_confirm: 1

Web API: NOT notified (to prevent orphaned incidents)



OPEN - Confirmed Incident

After 2+ consecutive detections, the incident is **confirmed**:

STATE: OPEN

What happened:

Anomaly persisted for 2+ detection cycles

What we do:

Send alert to web API

Track duration and occurrence count

Why confirm?

Persistent anomalies are likely real issues:

- Not a brief spike
- Sustained degradation
- Worth investigating

Payload fields:

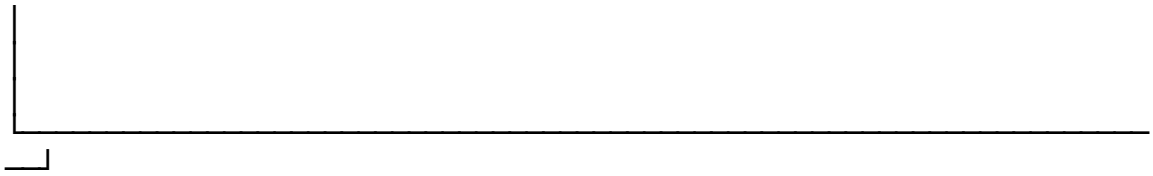
status: "OPEN"

is_confirmed: true

newly_confirmed: true (on first cycle only)

occurrence_count: incrementing

Web API: Alert sent!



RECOVERING - Grace Period

When the anomaly stops being detected, the incident enters a grace period:

STATE: RECOVERING

What happened:

Anomaly not detected in last cycle

What we do:

Wait before sending resolution

Watch for anomaly return

Why wait?

Brief recovery is often not real resolution:

- Issue may return in next cycle
- Prevents "flapping" alerts
- More reliable resolution signal

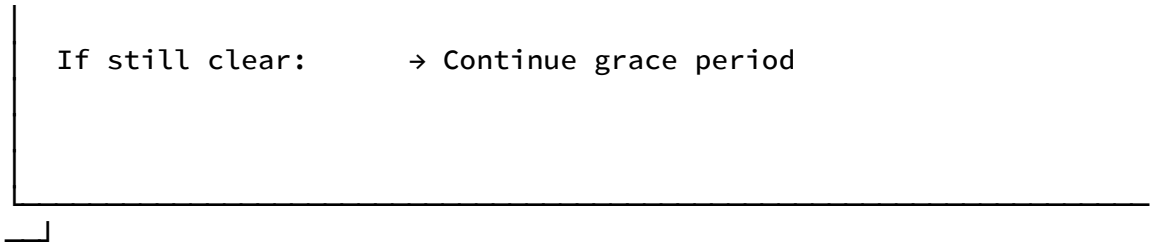
Payload fields:

status: "RECOVERING"

missed_cycles: 1, 2, ...

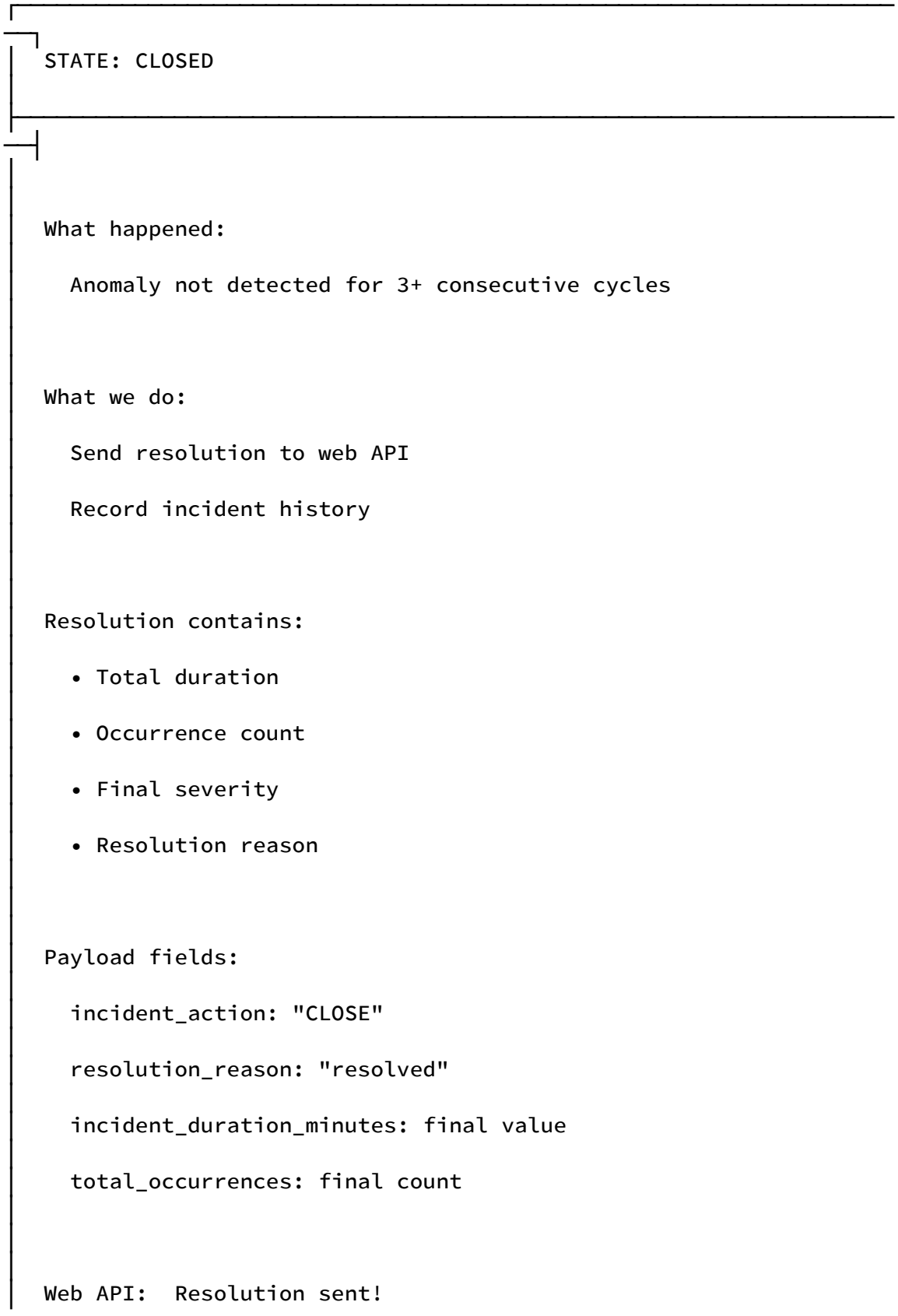
Web API: No update yet (waiting for confirmation)

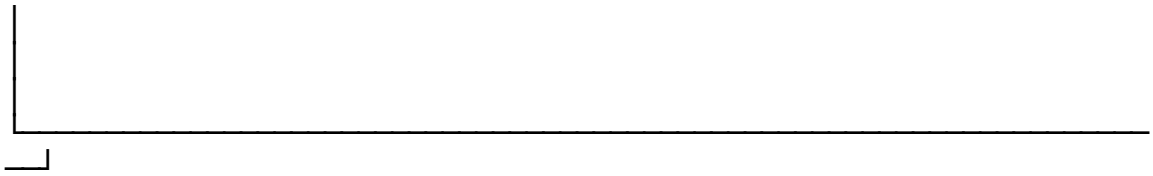
If anomaly returns: → Back to OPEN



CLOSED - Resolved

After 3+ cycles without detection, the incident is confirmed resolved:


STATE: CLOSED
What happened:
Anomaly not detected for 3+ consecutive cycles
What we do:
Send resolution to web API
Record incident history
Resolution contains:
• Total duration
• Occurrence count
• Final severity
• Resolution reason
Payload fields:
incident_action: "CLOSE"
resolution_reason: "resolved"
incident_duration_minutes: final value
total_occurrences: final count
Web API: Resolution sent!



Cycle-Based Timing

The lifecycle is based on **detection cycles**, not wall-clock time:

Parameter	Default	With 3-min inference	Purpose
`confirmation_cycles`	2	~6 min to confirm	Prevent false alerts
`resolution_grace_cycles`	3	~9 min grace period	Prevent flapping
`incident_separation_minutes`	30	30 min gap = new incident	Prevent zombie incidents

Example Timeline

```
Time     Detection   Status      Action
──────────────────────────────────────────────────
10:00    Spike!      SUSPECTED   Wait for confirmation
10:03    Spike!      OPEN        Alert sent (confirmed after 2
cycles)
10:06    Spike!      OPEN        Continue tracking
10:09    Spike!      OPEN        Continue tracking
10:12    Normal      RECOVERING  Grace period starts
10:15    Normal      RECOVERING  Grace period continues
10:18    Normal      CLOSED      Resolution sent (3 cycles without
detection)
```

Why Confirmation?

The problem: Single-detection alerts create noise.

```
WITHOUT CONFIRMATION:
─────────────────────
Every detection immediately fires an alert.

┌──────────────────────────────────────────────────────────────────┐
│  10:00  450ms latency detected  →  ALERT!  (page on-call)        │
│  10:03  120ms latency (normal)  →  Resolved                      │
│  10:06  455ms latency detected  →  ALERT!  (page on-call again)  │
│  10:09  118ms latency (normal)  →  Resolved                      │
│  10:12  460ms latency detected  →  ALERT!  (page on-call again)  │
│  10:15  115ms latency (normal)  →  Resolved                      │
└──────────────────────────────────────────────────────────────────┘

Result: 3 alerts for transient spikes
        On-call engineer interrupted 3 times
        Nothing was actually wrong
```

The solution: Require consecutive detections.

```
WITH CONFIRMATION (2 cycles):
─────────────────────────────

Only sustained anomalies fire alerts.

┌──────────────────────────────────────────────────────────────────┐
│  10:00  450ms latency detected  →  SUSPECTED (wait)              │
│  10:03  120ms latency (normal)  →  SUSPECTED expires quietly     │
│  10:06  455ms latency detected  →  SUSPECTED (wait)              │
│  10:09  460ms latency detected  →  OPEN - ALERT! (confirmed)     │
│  10:12  455ms latency detected  →  OPEN (continuing)             │
│  10:15  450ms latency detected  →  OPEN (continuing)             │
│  10:18  120ms latency (normal)  →  RECOVERING (grace period)     │
│  10:21  115ms latency (normal)  →  RECOVERING (grace period)     │
│  10:24  118ms latency (normal)  →  CLOSED - Resolved             │
└──────────────────────────────────────────────────────────────────┘

Result: 1 alert for a real sustained issue
        Resolution only after confirmed recovery
```

Why Grace Period?

The problem: Brief dips in anomaly cause flapping.

```
WITHOUT GRACE PERIOD:
─────────────────────

Every non-detection immediately resolves.

┌──────────────────────────────────────────────────────────────────┐
│  10:00  Error 5%  →  ALERT!                                      │
│  10:03  Error 5%  →  Continue                                    │
│  10:06  Error 0.5% →  Resolved!  (brief dip)                     │
│  10:09  Error 5%  →  ALERT!  (new incident)                      │
│  10:12  Error 5%  →  Continue                                    │
│  10:15  Error 0.5% →  Resolved!  (another brief dip)             │
│  10:18  Error 5%  →  ALERT!  (third incident)                    │
└──────────────────────────────────────────────────────────────────┘

Result: 3 separate incidents for what's really one ongoing issue
        "Flapping" alerts every time there's a brief improvement
```

The solution: Wait before confirming resolution.

```
WITH GRACE PERIOD (3 cycles):
─────────────────────────────

Resolution requires sustained recovery.

┌──────────────────────────────────────────────────────────────┐
│  10:00  Error 5%  →  ALERT!                                  │
│  10:03  Error 5%  →  Continue                                │
│  10:06  Error 0.5% →  RECOVERING (grace period, not resolved)│
│  10:09  Error 5%  →  OPEN (back to alerting - wasn't resolved)│
│  10:12  Error 5%  →  Continue                                │
│  10:15  Error 0.5% →  RECOVERING (grace period)              │
│  10:18  Error 0.5% →  RECOVERING (grace continues)           │
│  10:21  Error 0.5% →  CLOSED - Resolved (confirmed recovery) │
└──────────────────────────────────────────────────────────────┘

Result: 1 incident throughout entire episode
        Resolution only after confirmed sustained recovery
```

Staleness Check

The problem: Old incidents continuing forever.

If an anomaly is detected, disappears for hours, then reappears, is it the same incident?

```
10:00  - OPEN incident for latency_spike_recent
10:30  - Last detection
...
15:00  - Same pattern detected again (4.5 hours later!)

Is this the same incident? Probably not - it's a new occurrence.
```

The solution: `incident_separation_minutes` threshold.

```
If gap > incident_separation_minutes (default: 30 min):
  → Old incident auto-closed (reason: "auto_stale")
  → New SUSPECTED incident created

10:00  - OPEN incident created
10:30  - Last detection
...
11:15  - Same pattern detected (45 min gap > 30 min threshold)
         → Old incident CLOSED (auto_stale)
         → New SUSPECTED incident created
```

This prevents “zombie incidents” that span unrelated events.

Confirmed-Only Alerts (v1.3.2)

Important: Only confirmed anomalies are sent to the web API.

State	Sent to Web API?	Why?
SUSPECTED	**No**	Not yet confirmed - might be noise
OPEN	**Yes**	Confirmed - real incident
RECOVERING	**Yes**	Still tracking - might return
CLOSED	**Resolution only**	Final status update

Why Filter SUSPECTED?

Before v1.3.2, all detections were sent to the web API. This caused **orphaned incidents**:

```
OLD BEHAVIOR (problematic):
─────────────────────────────

10:00  SUSPECTED detected  →  Sent to API  →  API creates OPEN
incident
10:03  Not detected        →  SUSPECTED expires quietly
10:06  Not detected        →  (nothing sent)

Result: API has an OPEN incident that will never resolve
        "Orphaned incident" - no resolution was ever sent


NEW BEHAVIOR (v1.3.2):
──────────────────────

10:00  SUSPECTED detected  →  NOT sent to API (wait for
confirmation)
10:03  Not detected        →  SUSPECTED expires quietly

Result: API never knew about this transient detection
        No orphaned incidents
```

Summary Table

State	Alert Sent?	Resolution Sent?	Purpose
SUSPECTED	No	No	Wait for confirmation
OPEN	Yes	No	Active alerting
RECOVERING	No	No	Grace period
CLOSED	No	Yes	Final notification

Sections

- State Machine - Detailed state transitions and rules
- Confirmation Logic - How confirmation works
- Fingerprinting - How incidents are identified and tracked

State Machine

The incident state machine defines exactly how anomalies transition through the lifecycle. This page provides detailed rules, diagrams, and examples for each transition.

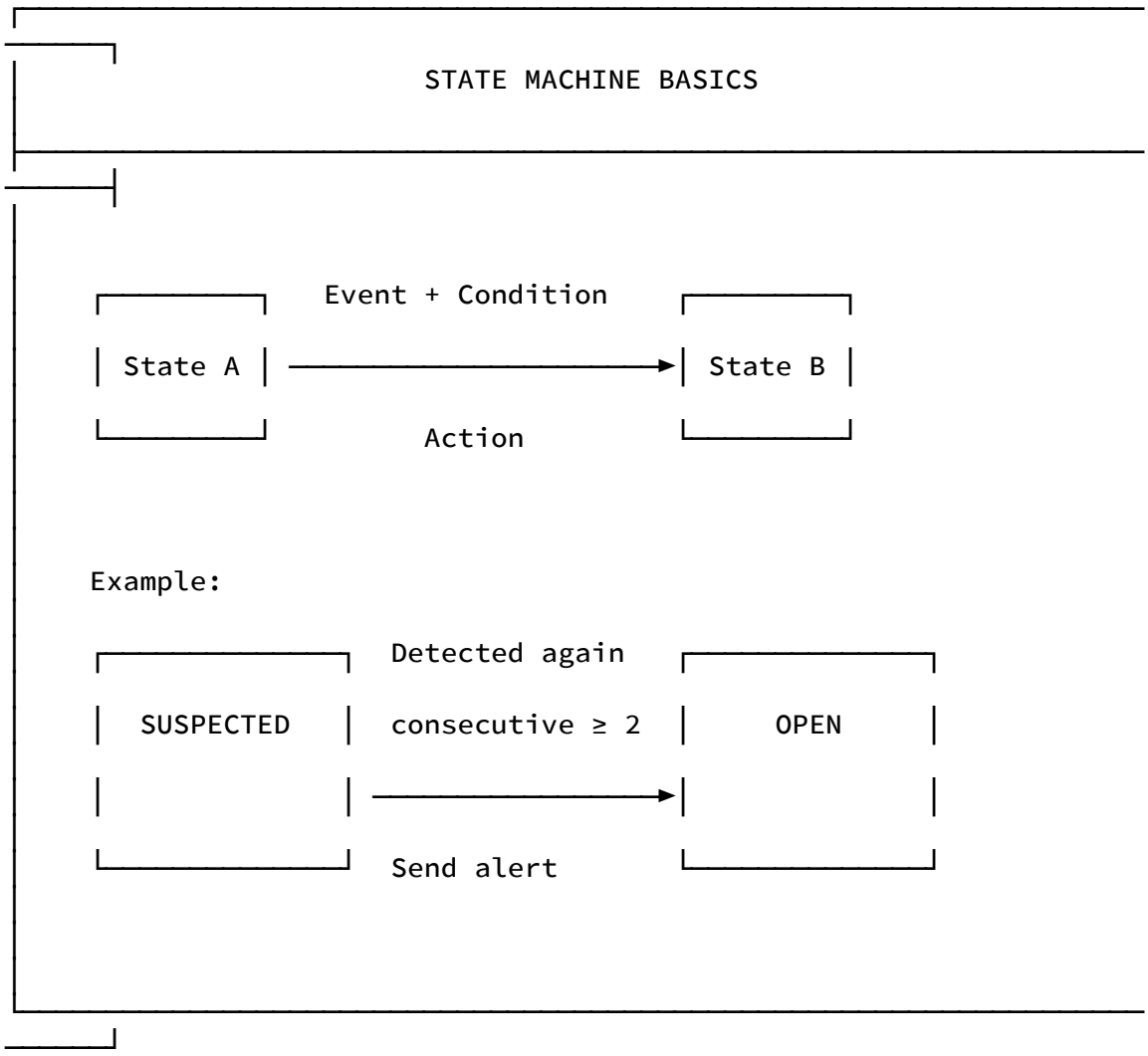
What is a State Machine?

A **state machine** is a model where:

- The system is always in exactly one **state**
- **Events** trigger transitions between states
- **Conditions** determine which transition occurs
- **Actions** execute when transitions happen

For incident tracking:

- **States**: SUSPECTED, OPEN, RECOVERING, CLOSED
- **Events**: “anomaly detected” or “anomaly not detected”
- **Conditions**: cycle counts, time gaps
- **Actions**: send alert, send resolution, update counters



Complete Transition Rules

This table defines all possible state transitions:

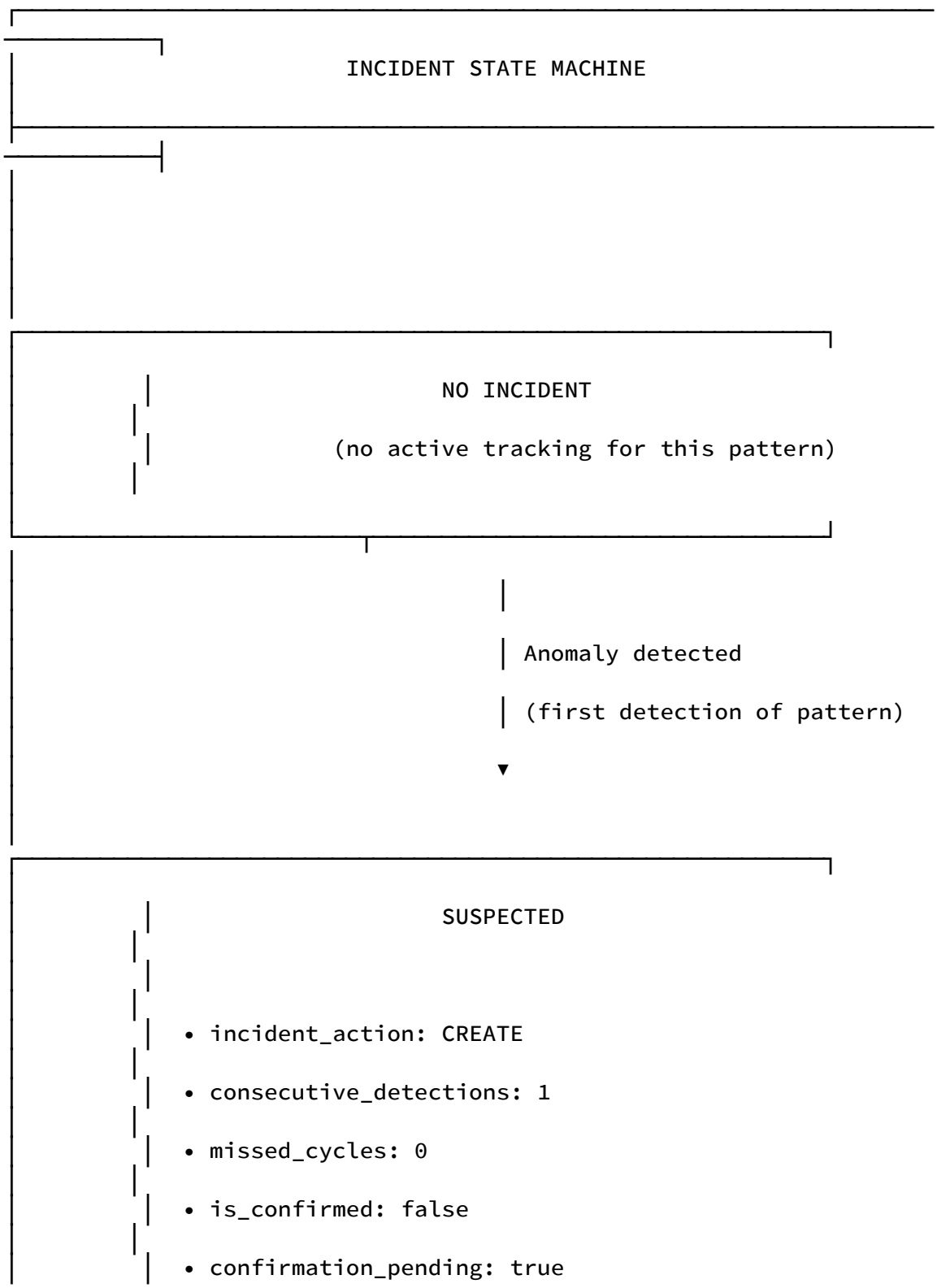
From State	Event	Condition	To State	
(none)	Detected	-	SUSPECTED	Create
SUSPECTED	Detected	`consecutive < N`	SUSPECTED	Increm `consec`

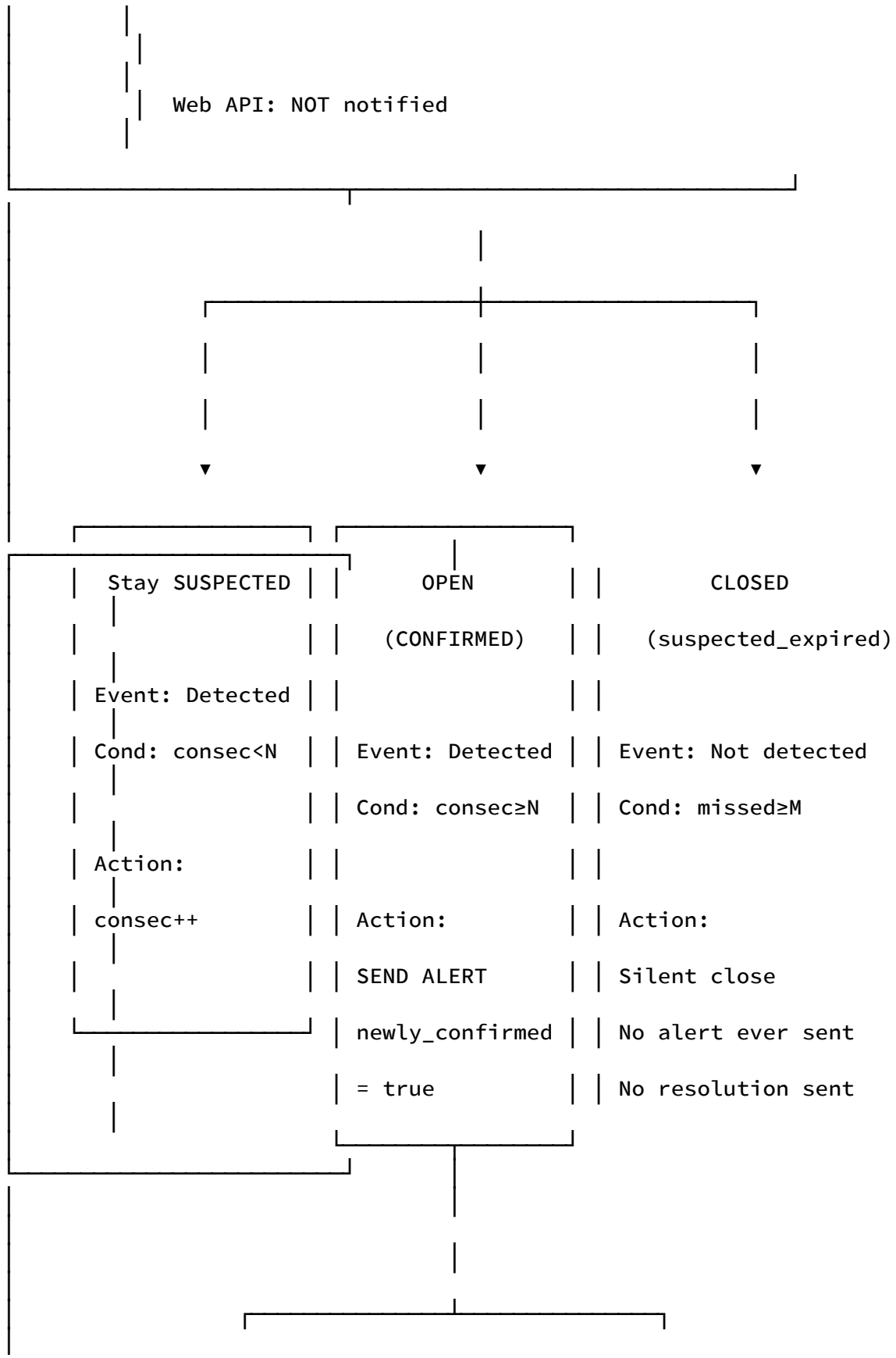
From State	Event	Condition	To State	
SUSPECTED	Detected	`consecutive ≥ N`	OPEN	**Send a**
SUSPECTED	Not detected	`missed < M`	SUSPECTED	Increm `misse`
SUSPECTED	Not detected	`missed ≥ M`	CLOSED	Silent (`suspe`
OPEN	Detected	-	OPEN	Contin `misse`
OPEN	Not detected	-	RECOVERING	Start g
RECOVERING	Detected	-	OPEN	Resum
RECOVERING	Not detected	`missed < M`	RECOVERING	Increm `misse`
RECOVERING	Not detected	`missed ≥ M`	CLOSED	**Send r**
(any)	Detected	`gap > threshold`	SUSPECTED	Close s

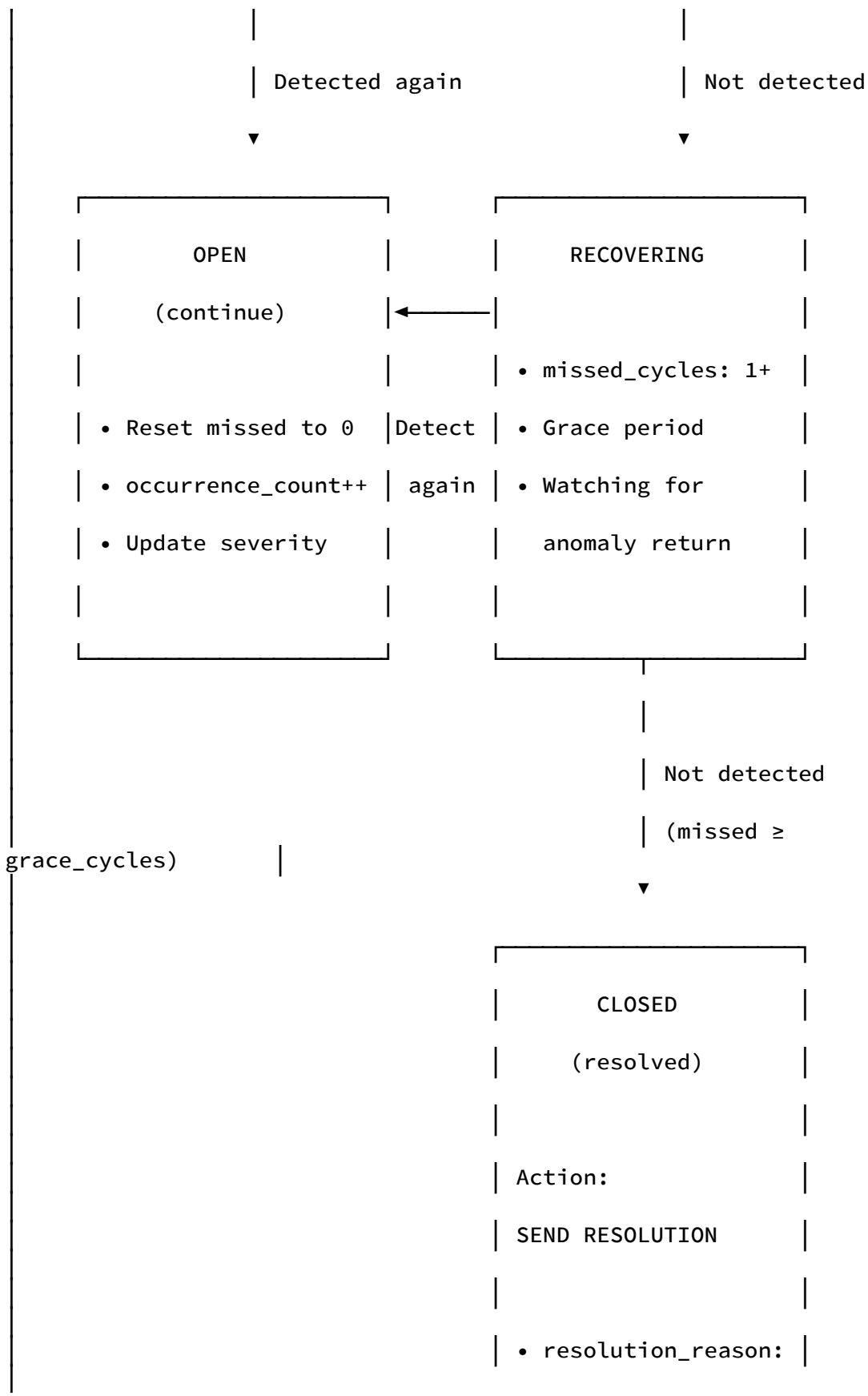
Where:

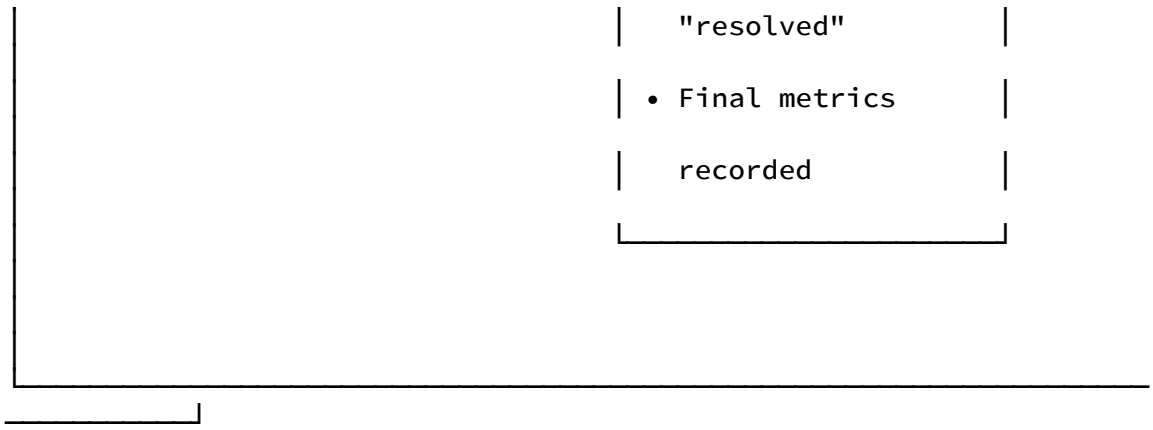
- `N` = `confirmation_cycles` (default: 2)
- `M` = `resolution_grace_cycles` (default: 3)
- `gap` = time since `last_updated`
- `threshold` = `incident_separation_minutes` (default: 30)

Visual State Diagram




Web API: NOT notified
Stay SUSPECTED
Event: Detected
Cond: consec<N
Action:
consec++
OPEN
(CONFIRMED)
Event: Detected
Cond: consec≥N
Action:
SEND ALERT
newly_confirmed
= true
CLOSED
(suspected_expired)
Event: Not detected
Cond: missed≥M
Action:
Silent close
No alert ever sent
No resolution sent


Detected again
Not detected
OPEN
(continue)
• Reset missed to 0
• occurrence_count++
• Update severity
Detect again
RECOVERING
• missed_cycles: 1+
• Grace period
• Watching for anomaly return
Not detected (missed ≥ grace_cycles)
CLOSED
(resolved)
Action:
SEND RESOLUTION
• resolution_reason:

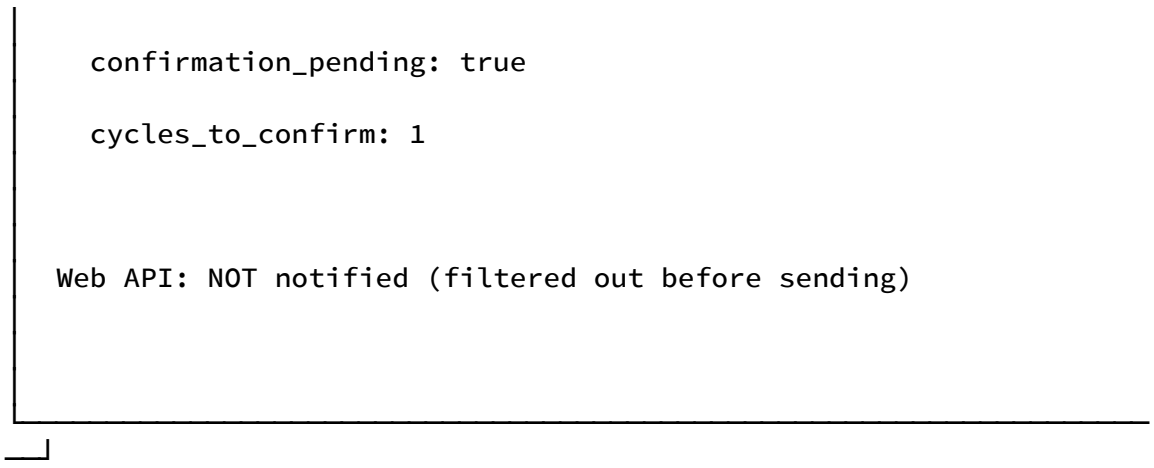


State-by-State Details

NO INCIDENT → SUSPECTED

Trigger: First detection of an anomaly pattern

```
Before:   No active incident for this fingerprint
Event:    ML detects anomaly_latency_spike_recent
After:    SUSPECTED incident created

┌──────────────────────────────────────────────────────────────────────────┐
│  Transition: NO INCIDENT → SUSPECTED                                     │
└──────────────────────────────────────────────────────────────────────────┘

  What happens:

    1. Generate fingerprint_id from pattern content

    2. Check database - no active incident exists

    3. Create new incident record

    4. Set status = SUSPECTED

    5. Initialize counters


  Payload fields set:

    fingerprint_id: "anomaly_abc123"  (deterministic hash)

    incident_id: "incident_xyz789"    (new UUID)

    fingerprint_action: "CREATE"

    incident_action: "CREATE"

    status: "SUSPECTED"

    consecutive_detections: 1

    missed_cycles: 0

    occurrence_count: 1

    first_seen: <current timestamp>

    is_confirmed: false
```



SUSPECTED → SUSPECTED (Still Waiting)

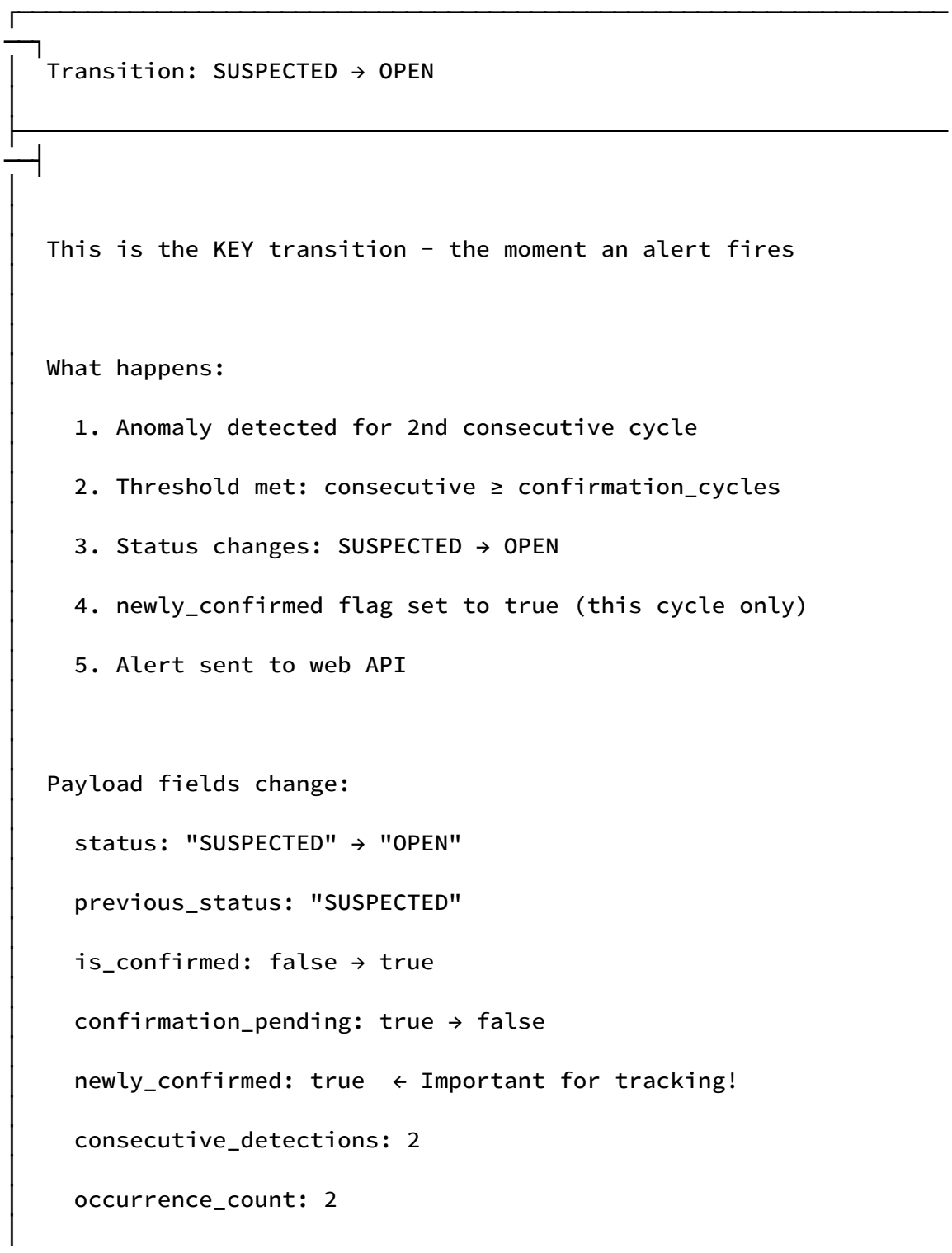
Trigger: Anomaly detected again, but not enough cycles yet

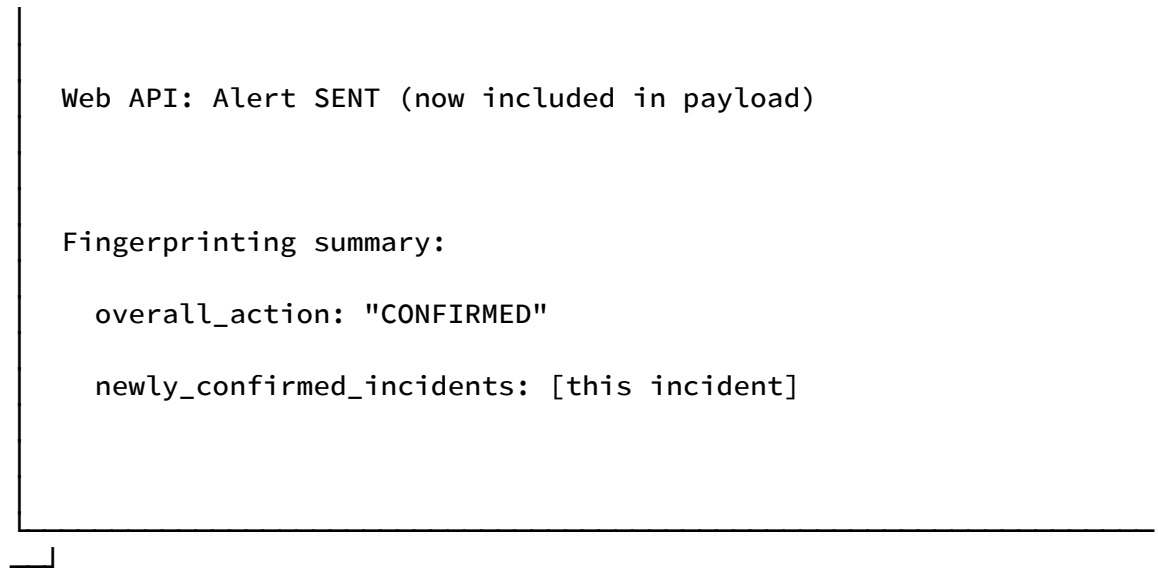
```
Before:   SUSPECTED with consecutive_detections = 1
Event:    Same anomaly detected
Cond:     consecutive_detections < confirmation_cycles
After:    SUSPECTED with consecutive_detections = 2

(With default confirmation_cycles=2, this transition goes to OPEN
instead)
```

SUSPECTED → OPEN (Confirmed!)

Trigger: Anomaly confirmed after N consecutive detections

```
Before:   SUSPECTED with consecutive_detections = 1
Event:    Same anomaly detected again
Cond:     consecutive_detections ≥ confirmation_cycles (default: 2)
After:    OPEN - alert sent!
```

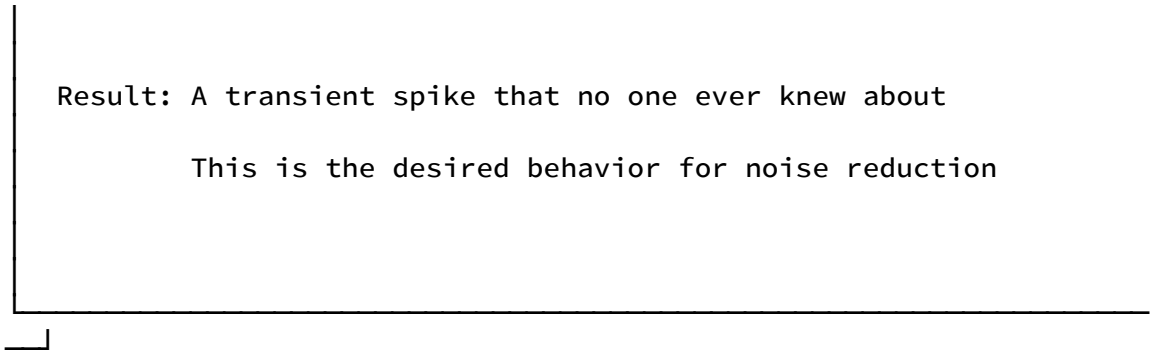




SUSPECTED → CLOSED (Expired Quietly)

Trigger: SUSPECTED anomaly not confirmed before grace period ends

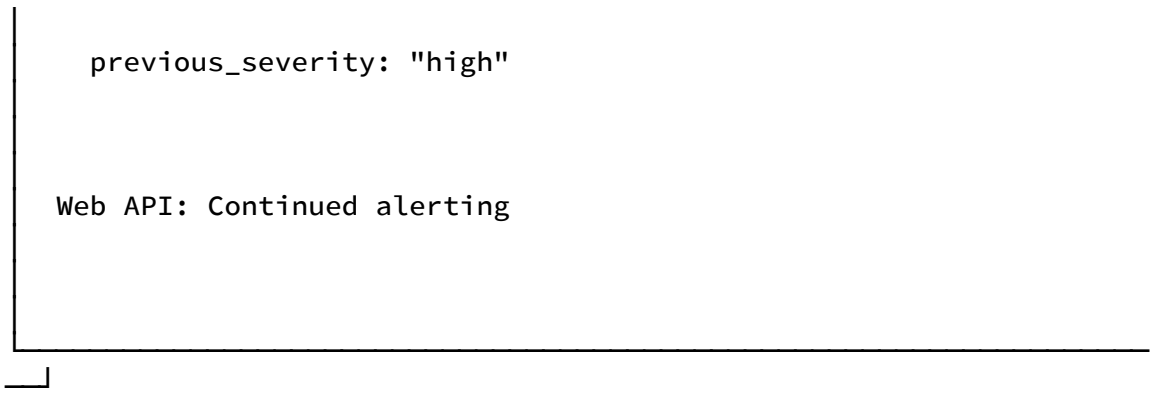
```
Before:   SUSPECTED with missed_cycles = 2
Event:    Anomaly NOT detected
Cond:     missed_cycles ≥ resolution_grace_cycles (default: 3)
After:    CLOSED with reason "suspected_expired"

┌──────────────────────────────────────────────────────────────────────┐
│  Transition: SUSPECTED → CLOSED (suspected_expired)                  │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│  This is a SILENT close - no alert was ever sent                     │
│                                                                      │
│  What happens:                                                       │
│    1. Anomaly detected once (SUSPECTED)                              │
│    2. Not detected in next 3 cycles                                  │
│    3. Incident closes without ever alerting                          │
│    4. No resolution sent to web API                                  │
│                                                                      │
│  Why no resolution?                                                  │
│    Web API was never notified of this incident.                      │
│    Sending a resolution would be confusing.                          │
│                                                                      │
│  Timeline example:                                                   │
│    10:00  Detected → SUSPECTED (not sent to API)                     │
│    10:03  Not detected → missed: 1                                   │
│    10:06  Not detected → missed: 2                                   │
│    10:09  Not detected → CLOSED (suspected_expired)                  │
└──────────────────────────────────────────────────────────────────────┘
```



OPEN → OPEN (Continue)

Trigger: Anomaly continues to be detected

```
Before:   OPEN with occurrence_count = 5
Event:    Same anomaly detected
After:    OPEN with occurrence_count = 6

┌──────────────────────────────────────────────────────────────────────────────┐
│  Transition: OPEN → OPEN (continue)                                          │
├──────────────────────────────────────────────────────────────────────────────┤
│                                                                              │
│  What happens:                                                               │
│    1. Anomaly still detected                                                 │
│    2. Counters updated                                                       │
│    3. missed_cycles reset to 0 (not in grace period)                         │
│    4. Duration updated                                                       │
│                                                                              │
│  Payload fields update:                                                      │
│    incident_action: "CONTINUE"                                               │
│    occurrence_count: ++                                                      │
│    consecutive_detections: ++                                                │
│    missed_cycles: 0 (reset)                                                  │
│    incident_duration_minutes: updated                                        │
│    last_updated: <current timestamp>                                         │
│                                                                              │
│  Severity can change:                                                        │
│    If anomaly severity changes (e.g., high → critical):                      │
│    severity: "high" → "critical"                                             │
│    severity_changed: true
```

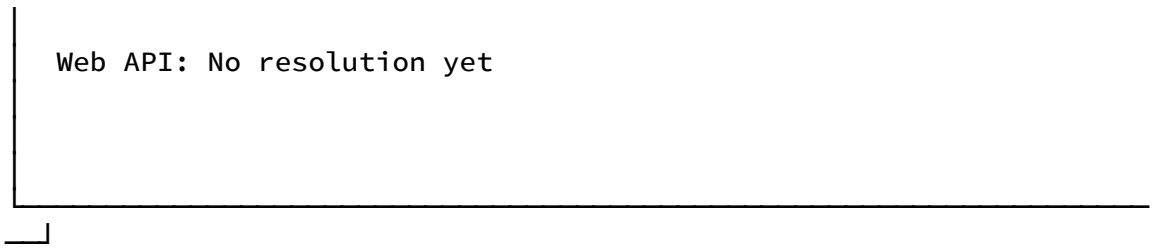


OPEN → RECOVERING

Trigger: Anomaly not detected, entering grace period

Before: OPEN
Event: Anomaly NOT detected
After: RECOVERING with missed_cycles = 1

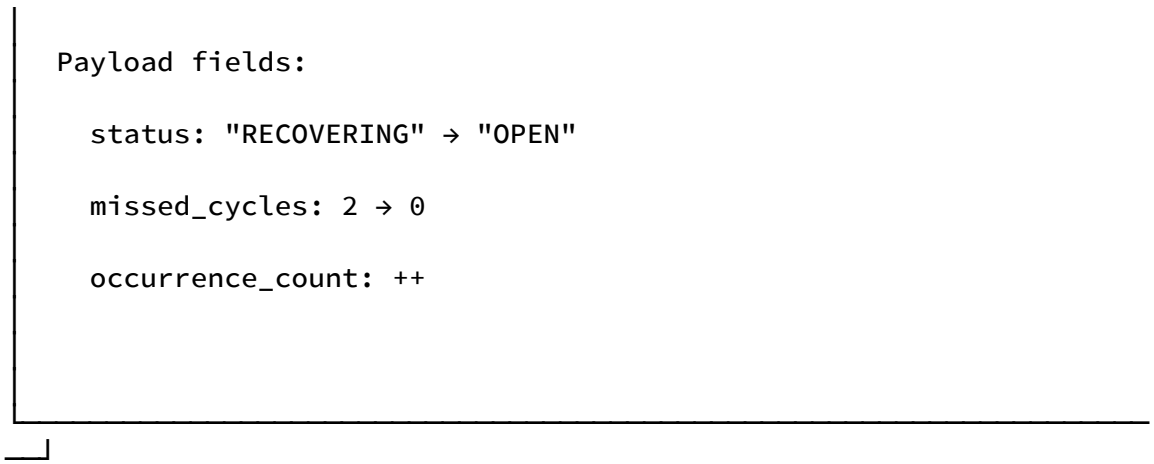
┌──┐
│ Transition: OPEN → RECOVERING   │
├──┤
│   │
│ What happens: │
│   1. Anomaly not detected this cycle   │
│   2. Enter grace period (don't close yet!)   │
│   3. Watch for anomaly return   │
│   │
│ Why grace period?   │
│   Anomalies often briefly clear before returning: │
│   • Brief improvement in latency   │
│   • One good measurement among bad   │
│   • Flapping behavior   │
│   │
│ Payload fields: │
│   status: "RECOVERING"   │
│   missed_cycles: 1   │
│   │
│ The fingerprinting summary shows: │
│   status_summary.recovering: 1   │
│   │



RECOVERING → OPEN (Anomaly Returns)

Trigger: Anomaly detected again during grace period

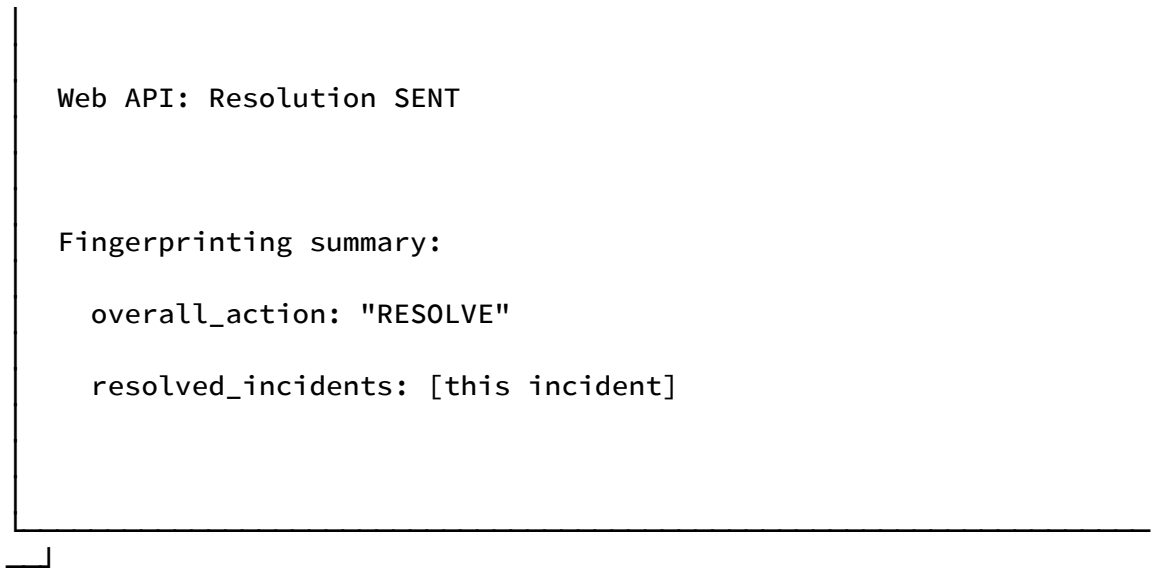
```
Before:   RECOVERING with missed_cycles = 2
Event:    Anomaly detected again
After:    OPEN - back to active alerting

┌──────────────────────────────────────────────────────────────────────┐
│  Transition: RECOVERING → OPEN                                       │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│  This is why grace periods exist!                                    │
│                                                                      │
│  What happens:                                                       │
│    1. Anomaly returns before grace period ends                       │
│    2. Resume tracking same incident                                  │
│    3. Reset missed_cycles to 0                                       │
│    4. Increment occurrence_count                                     │
│                                                                      │
│  Without grace period:                                               │
│    - Would have resolved after first miss                            │
│    - Would create NEW incident when anomaly returned                 │
│    - Same issue tracked as multiple incidents                        │
│                                                                      │
│  With grace period:                                                  │
│    - Brief clearance doesn't trigger resolution                      │
│    - Same incident continues when anomaly returns                    │
│    - Accurate duration tracking                                      │
│                                                                      │
```



RECOVERING → CLOSED (Resolved)

Trigger: Grace period completed without anomaly return

```
Before:   RECOVERING with missed_cycles = 2
Event:    Anomaly NOT detected
Cond:     missed_cycles ≥ resolution_grace_cycles (default: 3)
After:    CLOSED with reason "resolved"

┌──────────────────────────────────────────────────────────────────────────────┐
│  Transition: RECOVERING → CLOSED                                             │
├──────────────────────────────────────────────────────────────────────────────┤
│                                                                              │
│  What happens:                                                               │
│    1. Anomaly not detected for 3+ consecutive cycles                         │
│    2. Grace period completed                                                 │
│    3. Incident closed                                                        │
│    4. Resolution sent to web API                                             │
│                                                                              │
│  Resolution payload:                                                         │
│    fingerprint_id: "anomaly_abc123"                                          │
│    incident_id: "incident_xyz789"                                            │
│    fingerprint_action: "RESOLVE"                                             │
│    incident_action: "CLOSE"                                                  │
│    resolution_reason: "resolved"                                             │
│    final_severity: "high"                                                    │
│    resolved_at: <current timestamp>                                          │
│    total_occurrences: 8                                                      │
│    incident_duration_minutes: 45                                             │
│    first_seen: <original timestamp>                                          │
│                                                                              │
```



Staleness: Any → CLOSED + SUSPECTED

Trigger: Same pattern detected after long gap

```
Before:   OPEN (or RECOVERING) last updated 45 minutes ago
Event:    Anomaly detected
Cond:     gap > incident_separation_minutes (default: 30)
After:    Old incident CLOSED (auto_stale) + New SUSPECTED created
```

Staleness Transition

What happens:

1. Same anomaly pattern detected
2. But time gap exceeds threshold (30 min default)
3. Old incident auto-closed
4. New incident created from scratch

Why?

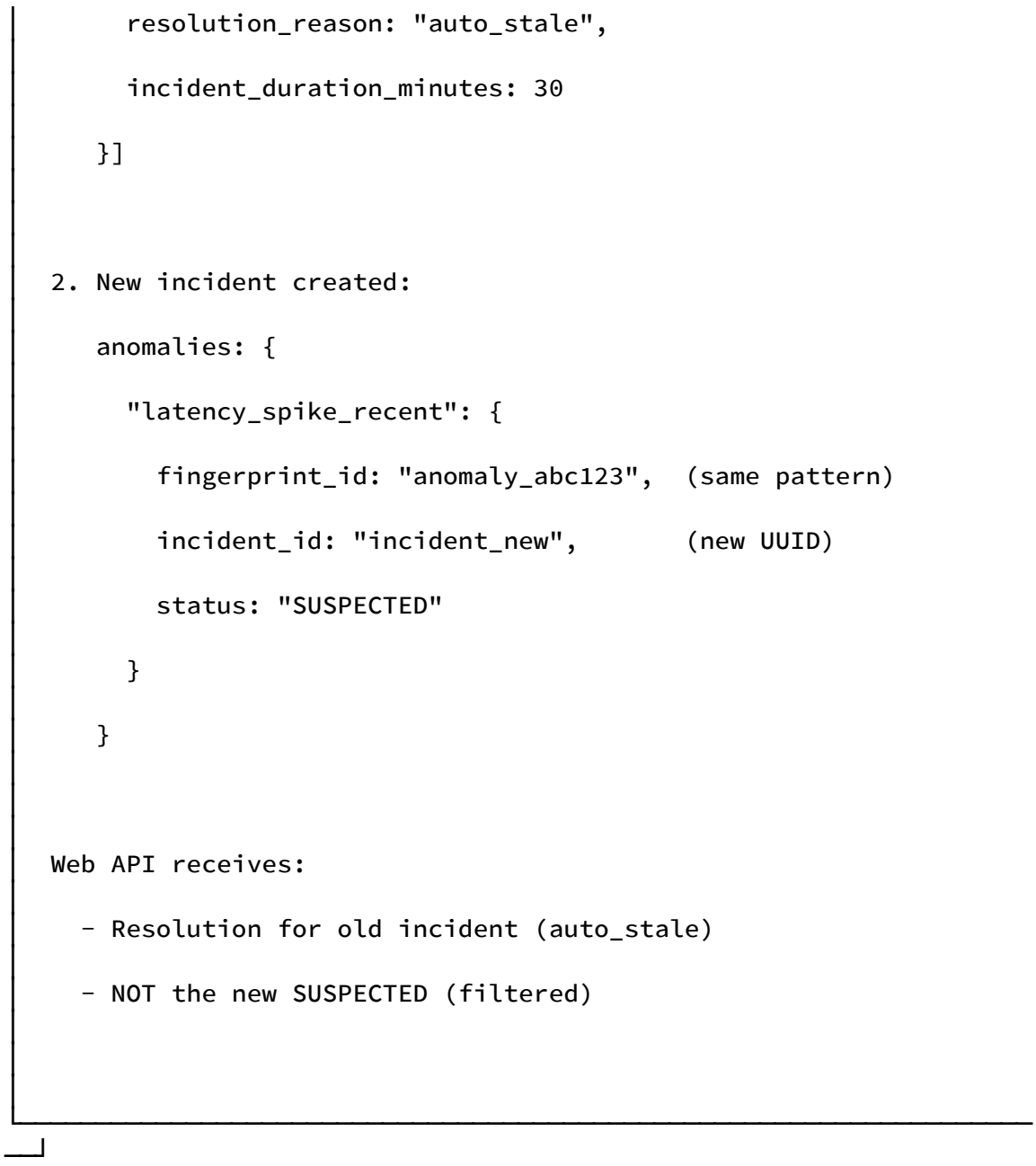
Long gaps usually indicate separate occurrences:

- Incident at 10:00, resolved by 10:30
- Same pattern at 15:00 - probably new issue
- Should be tracked separately

Two things happen in same cycle:

1. Old incident closed:

```
   resolved_incidents: [{
     incident_id: "incident_old",
```


resolution_reason: "auto_stale",
incident_duration_minutes: 30
}]
2. New incident created:
anomalies: {
"latency_spike_recent": {
fingerprint_id: "anomaly_abc123", (same pattern)
incident_id: "incident_new", (new UUID)
status: "SUSPECTED"
}
}
Web API receives:
- Resolution for old incident (auto_stale)
- NOT the new SUSPECTED (filtered)

Payload Fields by State

SUSPECTED State


```
{
  "fingerprint_id": "anomaly_d18f6ae2bf62",
  "incident_id": "incident_31e9e23d4b2b",
  "fingerprint_action": "CREATE",
  "incident_action": "CREATE",
  "status": "SUSPECTED",
  "consecutive_detections": 1,
  "missed_cycles": 0,
  "occurrence_count": 1,
  "first_seen": "2025-12-17T13:56:06.028585",
  "last_updated": "2025-12-17T13:56:06.028585",
  "incident_duration_minutes": 0,
  "confirmation_pending": true,
  "cycles_to_confirm": 1,
  "is_confirmed": false,
  "newly_confirmed": false
}
```

OPEN State (Newly Confirmed)


```
{
  "fingerprint_id": "anomaly_d18f6ae2bf62",
  "incident_id": "incident_31e9e23d4b2b",
  "fingerprint_action": "UPDATE",
  "incident_action": "CONTINUE",
  "status": "OPEN",
  "previous_status": "SUSPECTED",
  "consecutive_detections": 2,
  "missed_cycles": 0,
  "occurrence_count": 2,
  "first_seen": "2025-12-17T13:56:06.028585",
  "last_updated": "2025-12-17T13:59:06.028585",
  "incident_duration_minutes": 3,
  "confirmation_pending": false,
  "is_confirmed": true,
  "newly_confirmed": true
}
```


OPEN State (Continuing)


```
{
  "fingerprint_id": "anomaly_d18f6ae2bf62",
  "incident_id": "incident_31e9e23d4b2b",
  "fingerprint_action": "UPDATE",
  "incident_action": "CONTINUE",
  "status": "OPEN",
  "consecutive_detections": 5,
  "missed_cycles": 0,
  "occurrence_count": 5,
  "first_seen": "2025-12-17T13:56:06.028585",
  "last_updated": "2025-12-17T14:08:06.028585",
  "incident_duration_minutes": 12,
  "is_confirmed": true,
  "newly_confirmed": false
}
```

RECOVERING State

Note: RECOVERING incidents appear in `status_summary.recovering` count but individual anomalies are not in the payload (not detected this cycle).

Fingerprinting summary shows:


```
{
  "fingerprinting": {
    "overall_action": "NO_CHANGE",
    "status_summary": {
      "suspected": 0,
      "confirmed": 0,
      "recovering": 1
    }
  }
}
```


CLOSED State (Resolution)


```
{
  "resolved_incidents": [
    {
      "fingerprint_id": "anomaly_d18f6ae2bf62",
      "incident_id": "incident_31e9e23d4b2b",
      "anomaly_name": "latency_spike_recent",
      "fingerprint_action": "RESOLVE",
      "incident_action": "CLOSE",
      "final_severity": "high",
      "resolved_at": "2025-12-17T14:30:00.000000",
      "total_occurrences": 8,
      "incident_duration_minutes": 34,
      "first_seen": "2025-12-17T13:56:06.028585",
      "service_name": "booking",
      "last_detected_by_model": "business_hours",
      "resolution_reason": "resolved"
    }
  ]
}
```

Resolution Reasons

Reason	When	Alert Sent?	Resolution Sent?
`resolved`	Grace period completed	Yes (earlier)	Yes
`suspected_expired`	SUSPECTED never confirmed	No	No
`auto_stale`	Time gap exceeded threshold	Yes (earlier)	Yes

Configuration

```
{
  "fingerprinting": {
    "confirmation_cycles": 2,
    "resolution_grace_cycles": 3,
    "incident_separation_minutes": 30,
    "cleanup_max_age_hours": 72
  }
}
```

Parameter	Default	Range	Impact
`confirmation_cycles`	2	1-10	Higher = fewer false positives, more detection delay
`resolution_grace_cycles`	3	1-10	Higher = fewer flapping alerts,

Parameter	Default	Range	Impact
			longer resolution time
`incident_separation_minutes`	30	5-1440	Higher = more likely to continue existing incident
`cleanup_max_age_hours`	72	1-720	How long closed incidents stay in database

Complete Example: Full Lifecycle

```
Time     Event           State          Actions
──────────────────────────────────────────────────────────────────────
─
10:00    Detected        SUSPECTED      Create incident_abc, no
alert
10:03    Detected        OPEN           ALERT SENT! (confirmed)
10:06    Detected        OPEN           Continue tracking
10:09    Detected        OPEN           Continue tracking (count: 4)
10:12    Not detected    RECOVERING     Grace period starts (missed:
1)
10:15    Detected        OPEN           Resume incident (missed: 0)
10:18    Detected        OPEN           Continue tracking
10:21    Not detected    RECOVERING     Grace period (missed: 1)
10:24    Not detected    RECOVERING     Grace period (missed: 2)
10:27    Not detected    CLOSED         RESOLUTION SENT! (resolved)

Result:
  - 1 alert sent (at 10:03)
  - 1 resolution sent (at 10:27)
  - Duration: 27 minutes
  - Occurrences: 6
```

Confirmation Logic

Confirmation prevents alert noise by requiring multiple consecutive detections before alerting. This chapter explains why confirmation is essential, how it works, and how to tune it for your environment.

What is Confirmation?

Confirmation is the process of validating that an anomaly is real and persistent before triggering an alert. Instead of alerting immediately when something looks wrong, the system waits to see if the issue persists across multiple detection cycles.

```
Without Confirmation:          With Confirmation:

Detection 1 → ALERT!           Detection 1 → Wait...
(might be noise)               (could be transient)

Detection 2 → ALERT!           Detection 2 → ALERT!
(might be same issue)          (confirmed: real issue)
```

Think of it like a smoke detector: you don't want it to alarm for every wisp of steam from a shower. You want it to confirm there's actual smoke before waking everyone up.

Why Confirmation Matters

The Problem: Alert Fatigue

In production environments, metrics naturally fluctuate. A latency spike might last 30 seconds then disappear. An error rate might briefly increase during a

garbage collection pause. Without confirmation, every transient blip becomes an alert.

Without confirmation, operators experience:

- Multiple alerts per hour for transient issues
- “Resolved” notifications seconds after alerts
- Loss of trust in the alerting system
- Critical alerts buried in noise

Real-World Example

Consider a service during normal operation:

TIME	LATENCY	WITHOUT CONFIRMATION	WITH CONFIRMATION
10:00	120ms	Normal	Normal
10:03	450ms	🔴 ALERT: Latency!	⏳ SUSPECTED (1/2)
10:06	125ms	🟢 RESOLVED	⏳ Expires (no alert)
10:09	448ms	🔴 ALERT: Latency!	⏳ SUSPECTED (1/2)
10:12	455ms	(still alerting)	🔴 CONFIRMED (2/2)
10:15	460ms	(still alerting)	📊 Continue tracking
10:18	130ms	🟢 RESOLVED	⏳ Recovering (1/3)
10:21	128ms	Normal	⏳ Recovering (2/3)
10:24	125ms	Normal	🟢 RESOLVED

Results:

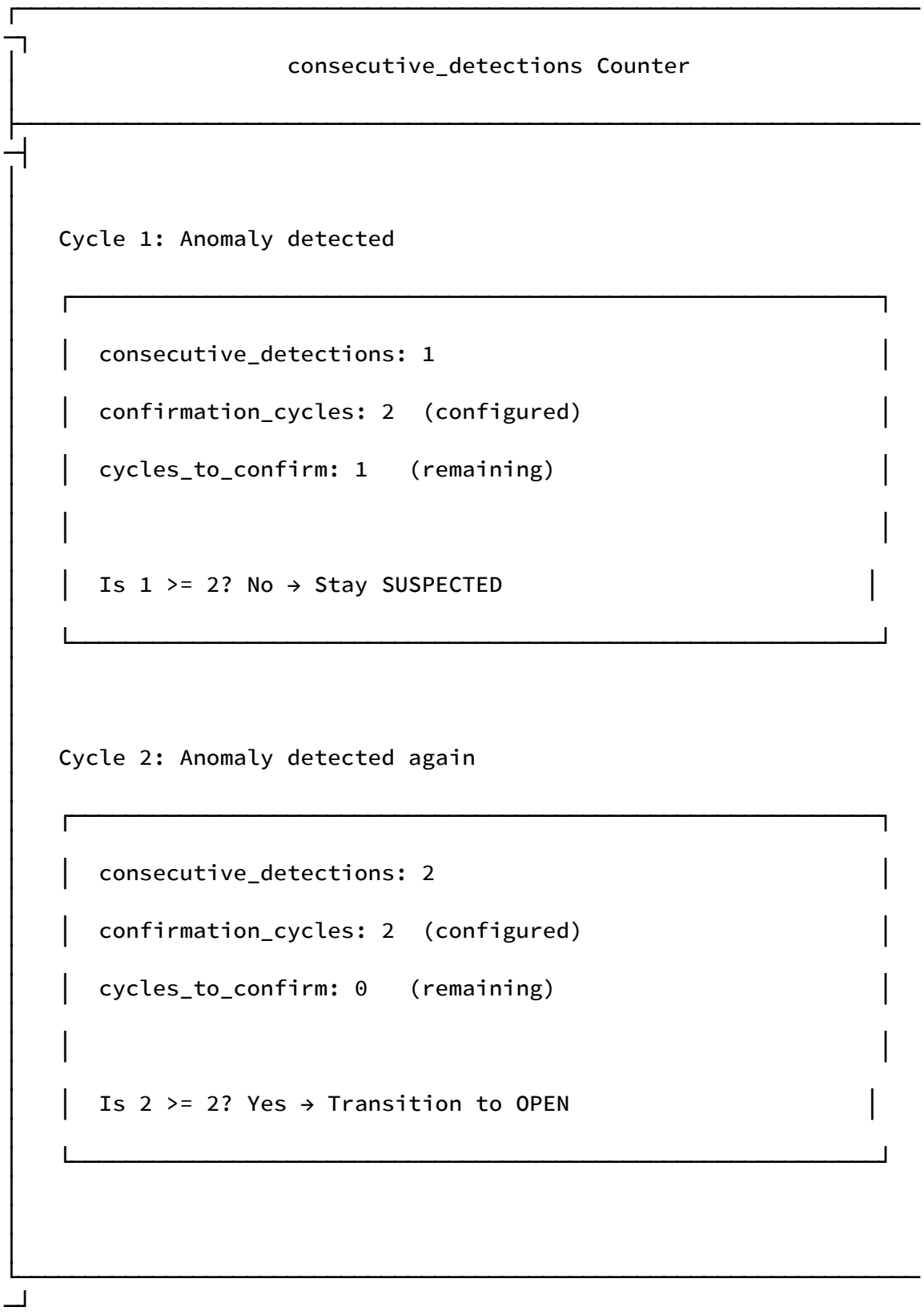
- Without confirmation: 2 alerts, 2 resolutions (noisy)
- With confirmation: 1 alert, 1 resolution (accurate)

The confirmed alert represents the real issue (10:09-10:15), while the transient spike at 10:03 was correctly filtered out.

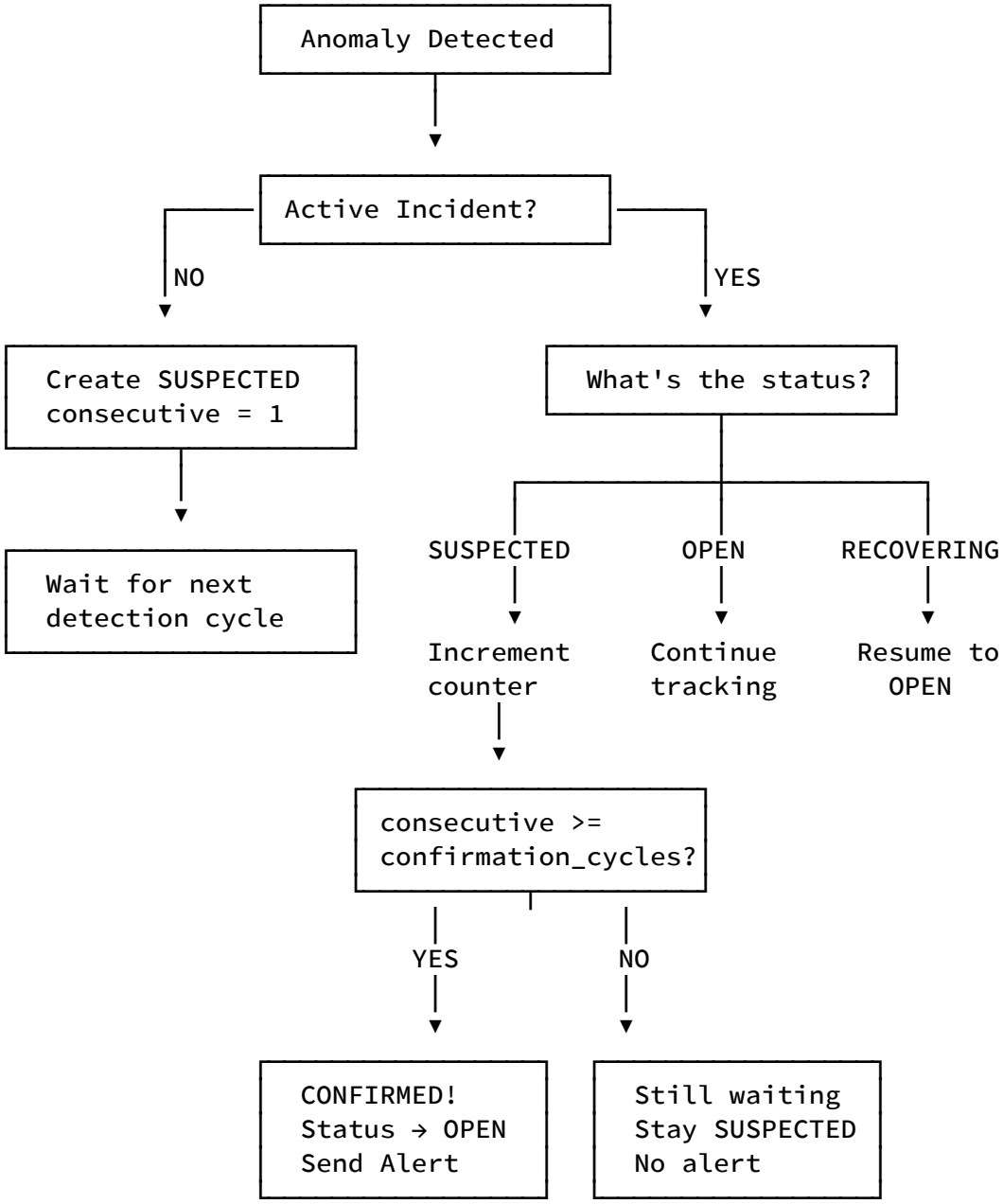
How Confirmation Works

The Confirmation Counter

Each incident tracks how many consecutive cycles it has been detected:


consecutive_detections Counter
Cycle 1: Anomaly detected
consecutive_detections: 1
confirmation_cycles: 2 (configured)
cycles_to_confirm: 1 (remaining)
Is 1 >= 2? No → Stay SUSPECTED
Cycle 2: Anomaly detected again
consecutive_detections: 2
confirmation_cycles: 2 (configured)
cycles_to_confirm: 0 (remaining)
Is 2 >= 2? Yes → Transition to OPEN

Confirmation Flow Diagram



Detection Cycles Explained

Cycle 1: First Detection (SUSPECTED)

When an anomaly is first detected, it enters the SUSPECTED state:

Detection Cycle 1

Time: 10:00:00

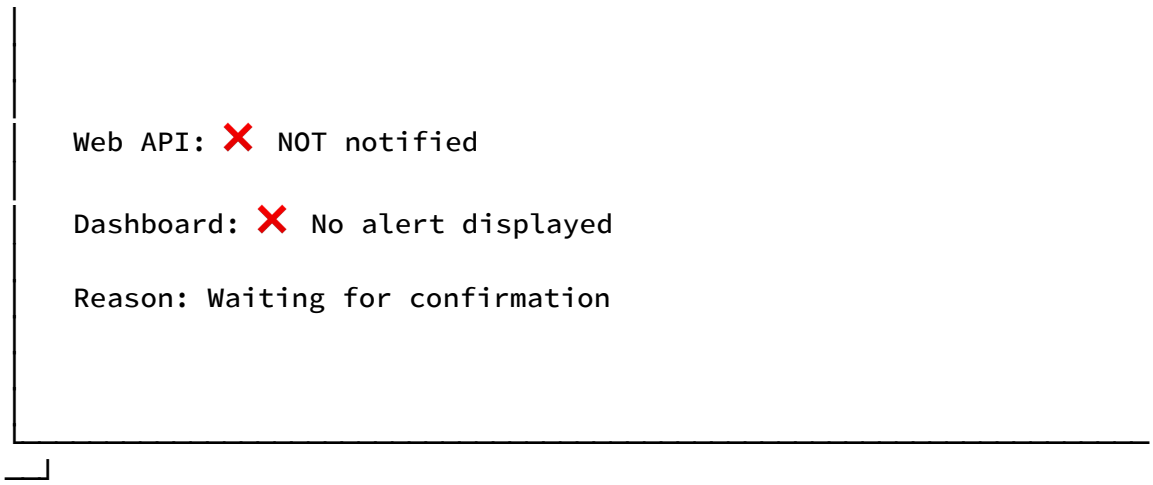
Event: Latency spike detected (450ms, normally 120ms)

Actions taken:

1. Generate fingerprint_id from pattern content
2. Check database for active incident with this fingerprint
3. No active incident found → Create new incident
4. Set status = SUSPECTED

Result:

```
fingerprint_id: anomaly_8d4a011b83ca
incident_id: incident_1dcbafc91480
status: SUSPECTED
consecutive_detections: 1
confirmation_pending: true
cycles_to_confirm: 1
is_confirmed: false
```



Cycle 2: Confirmation (SUSPECTED → OPEN)

If the same anomaly is detected again in the next cycle:

Detection Cycle 2

Time: 10:03:00 (3 minutes later)

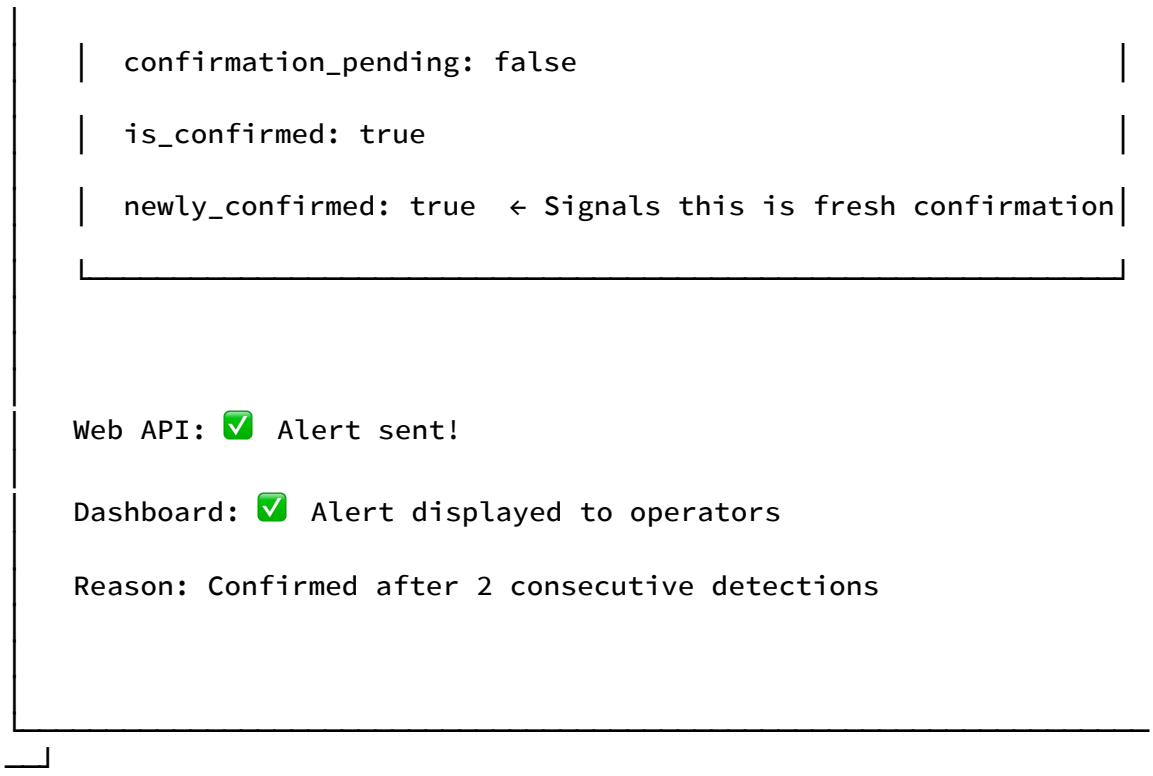
Event: Same latency pattern detected again

Actions taken:

1. Generate fingerprint_id from pattern content
2. Check database → Found existing SUSPECTED incident
3. Increment consecutive_detections: 1 → 2
4. Check: 2 >= confirmation_cycles (2)? YES!
5. Transition status: SUSPECTED → OPEN
6. Set newly_confirmed = true
7. Send alert to Web API

Result:

```
fingerprint_id: anomaly_8d4a011b83ca
incident_id: incident_1dcbafc91480
status: OPEN
previous_status: SUSPECTED  ← For tracking the transition
consecutive_detections: 2
```



The `newly_confirmed` Flag

The `newly_confirmed` flag is crucial for downstream consumers:

```
When newly_confirmed = true:
```

- This is the FIRST time this incident is being sent as confirmed
- Web UI should create a new alert entry
- Notification systems should send alerts
- Only set on the exact cycle of confirmation

```
When newly_confirmed = false:
```

- Incident was already confirmed in a previous cycle
- Web UI should update existing alert entry
- Notification systems should NOT re-alert
- Set for all subsequent cycles

Key Fields Reference

Field	Type	Description
`consecutive_detections`	integer	How many cycles in a row this anomaly was detected
`confirmation_pending`	boolean	`true` while still in SUSPECTED state
`cycles_to_confirm`	integer	Remaining cycles needed for confirmation (0 when confirmed)
`is_confirmed`	boolean	`true` once status becomes OPEN
`newly_confirmed`	boolean	`true` only on the exact cycle where SUSPECTED → OPEN
`previous_status`	string	What the status was before this cycle (for tracking transitions)

Fingerprinting Summary Object

The top-level `fingerprinting` object provides a summary of all confirmation activity:

```
{
  "fingerprinting": {
    "service_name": "booking",
    "model_name": "business_hours",
    "timestamp": "2025-12-17T10:03:00",
    "overall_action": "CONFIRMED",

    "status_summary": {
      "suspected": 0,
      "confirmed": 1,
      "recovering": 0
    },

    "action_summary": {
      "incident_creates": 0,
      "incident_continues": 0,
      "incident_closes": 0,
      "newly_confirmed": 1
    },

    "newly_confirmed_incidents": [
      {
        "fingerprint_id": "anomaly_8d4a011b83ca",
        "incident_id": "incident_1dcbafc91480",
        "anomaly_name": "latency_spike_recent",
        "severity": "high"
      }
    ]
  }
}
```


Overall Action Values

Action	Meaning	When It Happens
`CREATE`	New incident(s) created in SUSPECTED state	First detection of new anomaly patterns
`CONFIRMED`	Incident(s) transitioned SUSPECTED → OPEN	Anomaly detected for confirmation_cycles times

Action	Meaning	When It Happens
`UPDATE`	Existing OPEN incident(s) continued	Ongoing anomaly still being detected
`RESOLVE`	Incident(s) closed	Grace period exceeded without detection
`MIXED`	Multiple different actions in one cycle	E.g., one confirmed while another closes
`NO_CHANGE`	No significant state changes	Only RECOVERING incidents still waiting

Web API Integration

Confirmed-Only Alerts (v1.3.2)

Starting with version 1.3.2, only **confirmed** anomalies are sent to the web API. This is a critical feature for preventing orphaned incidents.

```
# How the inference engine filters before sending to web API

def process_results(anomalies):
    # Filter to only confirmed anomalies
    confirmed_anomalies = {
        name: anomaly for name, anomaly in anomalies.items()
        if anomaly.get('is_confirmed', False) or
           anomaly.get('status') in ('OPEN', 'RECOVERING')
    }

    if confirmed_anomalies:
        # Send only confirmed anomalies to web API
        send_alert(confirmed_anomalies)
    else:
        # SUSPECTED anomalies are NOT sent
        # This prevents orphaned incidents
        pass
```

Why This Matters

Before v1.3.2, all anomalies (including SUSPECTED) were sent to the web API. This caused problems:

```
OLD BEHAVIOR (problematic):
─────────────────────────────

10:00  Detection → SUSPECTED → Sent to Web API → Web API creates
OPEN incident
10:03  Not detected → SUSPECTED expires → No resolution sent
(suspected_expired)
10:06  ...
Result: Orphaned OPEN incident in Web API that never gets resolved!

NEW BEHAVIOR (correct):
─────────────────────────

10:00  Detection → SUSPECTED → NOT sent to Web API (waiting for
confirmation)
10:03  Not detected → SUSPECTED expires → Nothing to resolve (never
sent)
10:06  ...
Result: No orphaned incident – Web API never knew about it!

OR if it gets confirmed:

10:00  Detection → SUSPECTED → NOT sent to Web API
10:03  Detection → OPEN (confirmed) → Sent to Web API → Web API
creates incident
10:06  Detection → OPEN continues → Update sent
...
10:15  Not detected × 3 cycles → CLOSED → Resolution sent
Result: Complete lifecycle – incident created when confirmed,
resolved when cleared
```

SUSPECTED Expiration

When an anomaly is detected but then disappears before confirmation:

```
Timeline of SUSPECTED Expiration:
──────────────────────────────────

10:00:00  Anomaly detected
          ├─ Status: SUSPECTED
          ├─ consecutive_detections: 1
          └─ Web API: NOT notified

10:03:00  Anomaly NOT detected
          ├─ Status: still SUSPECTED
          ├─ missed_cycles: 1
          └─ Web API: still not notified

10:06:00  Anomaly NOT detected
          ├─ Status: still SUSPECTED
          ├─ missed_cycles: 2
          └─ Web API: still not notified

10:09:00  Anomaly NOT detected
          ├─ missed_cycles: 3 >= resolution_grace_cycles
          ├─ Status: SUSPECTED → CLOSED
          ├─ resolution_reason: "suspected_expired"
          ├─ Web API: NO resolution sent (never was an incident
there)
          └─ Incident removed from tracking

Result:
─────────
```

- No alert was ever sent
- No resolution is needed
- Transient issue correctly filtered
- Zero noise in the alerting system

Resolution Reasons for SUSPECTED

Reason	Description	Web API Impact
`suspected_expired`	Never confirmed, disappeared before confirmation	Nothing sent (no orphan)
`resolved`	Normal resolution after grace period	N/A (only applies to

Reason	Description	Web API Impact
		OPEN)
`auto_stale`	Time gap exceeded threshold	N/A (only applies to OPEN)

Tuning Confirmation Cycles

The `confirmation_cycles` configuration determines how many consecutive detections are required:

```
{
  "fingerprinting": {
    "confirmation_cycles": 2
  }
}
```

Trade-off Analysis

Value	Confirmation Time*	Pros	Cons
1	Immediate	Fastest response	No filtering, all noise
2	~4-6 min	**Good balance (default)**	1 cycle delay
3	~6-9 min	Fewer false positives	May miss short incidents
4	~8-12 min	Very strict filtering	Risk of missing real issues
5+	10+ min	Maximum noise reduction	Likely too slow for production

*Assuming 2-3 minute detection cycles

Choosing the Right Value

Use `confirmation_cycles: 1` when:

- Testing or debugging the system
- You need immediate alerts regardless of noise
- You have other mechanisms to filter alerts downstream

Use `confirmation_cycles: 2` (default) when:

- Running in production
- You want balanced noise reduction
- Detection cycle is 2-5 minutes

Use `confirmation_cycles: 3` when:

- You have a noisy environment with frequent transient issues
- False positives are more costly than delayed detection
- You can tolerate 6-9 minute confirmation delay

Use `confirmation_cycles: 4+` when:

- You have very long detection cycles (10+ minutes)
- You're monitoring non-critical services
- Alert fatigue is a severe problem

Edge Cases

Pattern Changes During Confirmation

If the anomaly pattern changes during confirmation, it's treated as a new anomaly:

```
10:00  latency_spike_recent detected → SUSPECTED
10:03  traffic_surge_failing detected → NEW SUSPECTED (different
pattern)
       (latency_spike_recent starts expiration countdown)
```

Multiple Anomalies Confirming Simultaneously

Multiple anomalies can confirm in the same cycle:

```
{
  "fingerprinting": {
    "overall_action": "MIXED",
    "newly_confirmed_incidents": [
      {"anomaly_name": "latency_spike_recent", ...},
      {"anomaly_name": "error_rate_elevated", ...}
    ],
    "action_summary": {
      "newly_confirmed": 2
    }
  }
}
```

Confirmation After Recovery

If an incident is in RECOVERING state and the anomaly reappears, it doesn't need re-confirmation:

```
10:00  SUSPECTED (1/2)
10:03  OPEN (confirmed)
10:06  OPEN (continuing)
10:09  RECOVERING (not detected, 1/3)
10:12  OPEN (detected again – immediately returns to OPEN, no
confirmation needed)
```

This is because the incident was already confirmed before entering RECOVERING.

Summary

Confirmation is a critical noise-reduction mechanism that:

1. **Prevents alert fatigue** by filtering transient spikes
2. **Ensures reliability** by only alerting on persistent issues
3. **Protects the web API** from orphaned incidents (confirmed-only alerts)
4. **Provides clear lifecycle** with trackable state transitions

The default configuration (`confirmation_cycles: 2`) provides a good balance between responsiveness and noise reduction for most production environments.

Fingerprinting

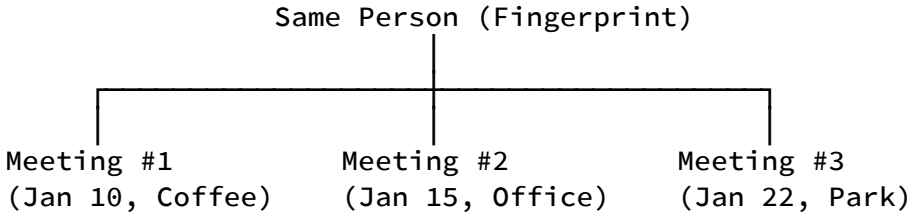
Fingerprinting assigns stable identifiers to anomaly patterns, enabling tracking across detection cycles. This chapter explains the identification system, how it works, and why two different types of IDs are needed.

What is Fingerprinting?

Fingerprinting is the process of creating stable, reproducible identifiers for anomaly patterns. When the system detects an anomaly, it generates a "fingerprint" - a unique identifier based on what the anomaly is, not when it happened.

Think of it like identifying a person:

- **Fingerprint** = A person's actual fingerprint (uniquely identifies the person)
- **Incident** = A specific encounter with that person (when and where you met them)



Each meeting is a unique incident, but it's the same person.

Why Two ID Types?

The system uses two different identifiers that serve complementary purposes:

ID Type	What It Identifies	Characteristics
Fingerprint ID	The anomaly pattern itself	Stable, deterministic, content-based
Incident ID	A specific occurrence of the pattern	Unique, random, time-bound

Why Not Just Use One ID?

Using only one ID would create problems:

If only fingerprint_id:

```
Problem: Can't distinguish between separate occurrences

Jan 10  latency_spike (fingerprint_abc) - Incident A
Jan 15  latency_spike (fingerprint_abc) - Is this Incident A
continuing?
                                          Or a new incident?
                                          We can't tell!
```

If only incident_id:

```
Problem: Can't tell if issues are related

Jan 10  incident_123 (latency spike)
Jan 15  incident_456 (latency spike) - Are these the same type of
issue?
                                       Different issue types?
                                       We can't tell!
```

With both IDs:

```
Jan 10  fingerprint_abc / incident_123 (latency spike)
Jan 15  fingerprint_abc / incident_456 (latency spike)

Now we know:
- Same TYPE of anomaly (same fingerprint_abc)
- Different OCCURRENCES (different incident IDs)
- We can track patterns AND individual events
```

Fingerprint ID: Pattern Identity

What Is It?

A **deterministic hash** based on the anomaly's content - what it is, not when it happened.

How It's Generated

```
# The fingerprint is generated from these components:
content = f"{service_name}_{model_name}_{anomaly_name}"
fingerprint_id = f"anomaly_{sha256(content)[:12]}"
```

Example:

```
Components:
  service_name: booking
  model_name: business_hours
  anomaly_name: latency_spike_recent

Concatenated: "booking_business_hours_latency_spike_recent"
SHA256 hash:  d18f6ae2bf62a7c9...
Fingerprint:  anomaly_d18f6ae2bf62

Same inputs ALWAYS produce the same fingerprint!
```

Properties

Property	Description	Example
Deterministic	Same pattern always gets same ID	`booking_business_hours_latency_spi` → always `anomaly_d18f6ae2bf62`
Content-based	Based on what, not	Doesn't include timestamp

Property	Description	Example
	when	
Stable	Doesn't change across time	Same ID today, tomorrow, next year
Reproducible	Can be regenerated	Given the same inputs, always same ou

What Changes the Fingerprint?

The fingerprint changes when the pattern type changes:

```
SAME fingerprint (anomaly_abc):
──────────────────────────────
• booking + business_hours + latency_spike_recent (Monday)
• booking + business_hours + latency_spike_recent (Tuesday)
• booking + business_hours + latency_spike_recent (Different
severity)

DIFFERENT fingerprint (anomaly_xyz):
────────────────────────────────────
• booking + business_hours + traffic_surge_failing (different
anomaly type)
• booking + evening_hours + latency_spike_recent (different time
period)
• search + business_hours + latency_spike_recent (different service)
```

Incident ID: Occurrence Identity

What Is It?

A **unique identifier** for each specific occurrence of an anomaly pattern.

How It's Generated

```
# The incident ID is generated randomly when a new incident is
created:
incident_id = f"incident_{uuid4().hex[:16]}"
```

Example:

```
Each new incident gets a new random ID:
  incident_1dcbafc91480
  incident_7a23b9f4c8e1
  incident_95d2e61a8b43

Even for the same pattern type, each occurrence has a unique ID!
```

Properties

Property	Description	Example
Unique	Each occurrence gets its own ID	No two incidents share an ID
UUID-based	Random generation	Not predictable from inputs
Transient	New ID when pattern reappears	After resolution, next occurrence = new ID
Time-bound	Associated with specific time period	Tracks one continuous occurrence

When Is a New Incident ID Created?

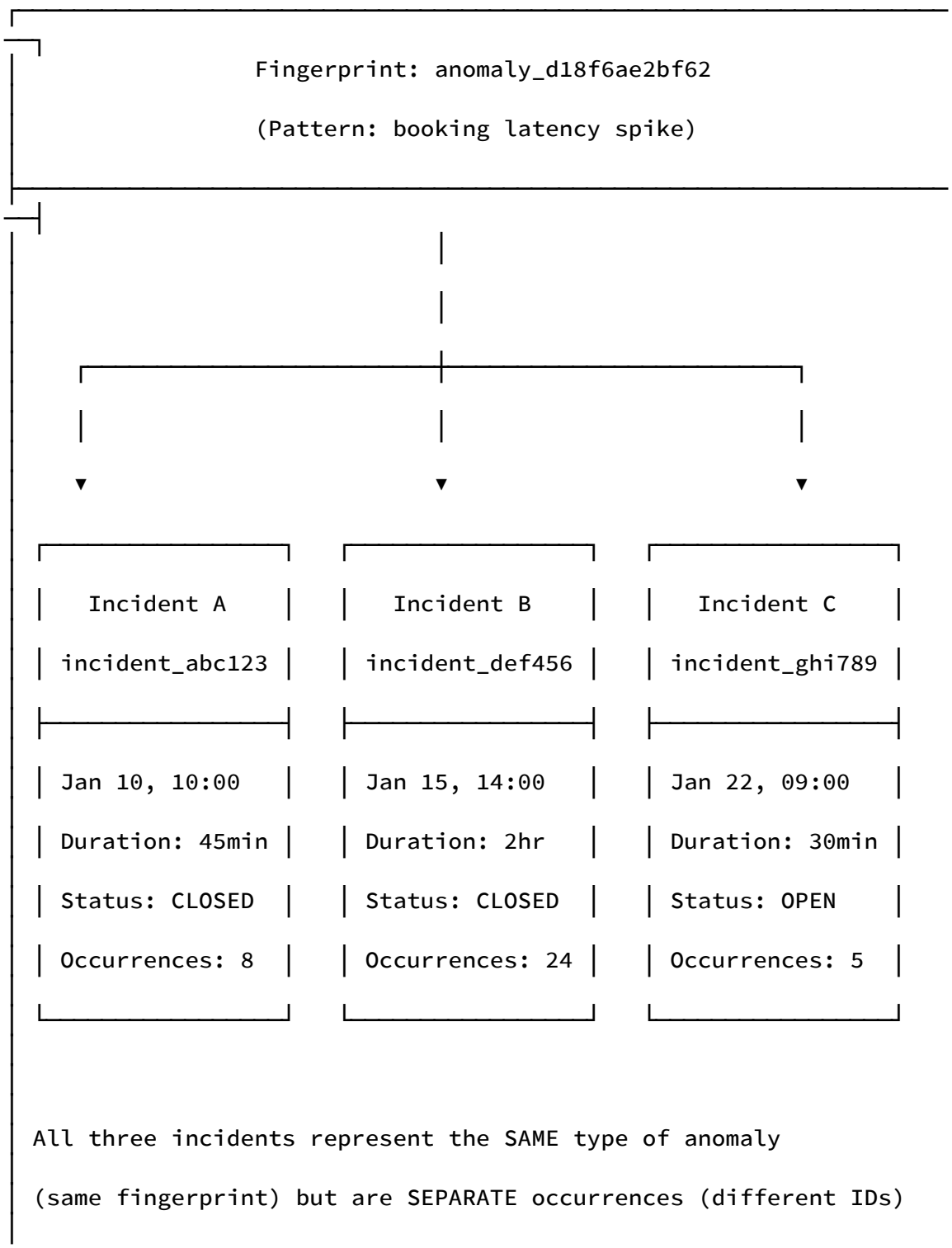
New incident_id created when:

- Anomaly detected for first time (no active incident)
- Anomaly reappears after being resolved
- Anomaly reappears after staleness threshold (>30 min gap)

Same incident_id continues when:

- Anomaly detected while incident is OPEN
- Anomaly detected while incident is RECOVERING
- Anomaly detected within staleness threshold

The Relationship Between IDs



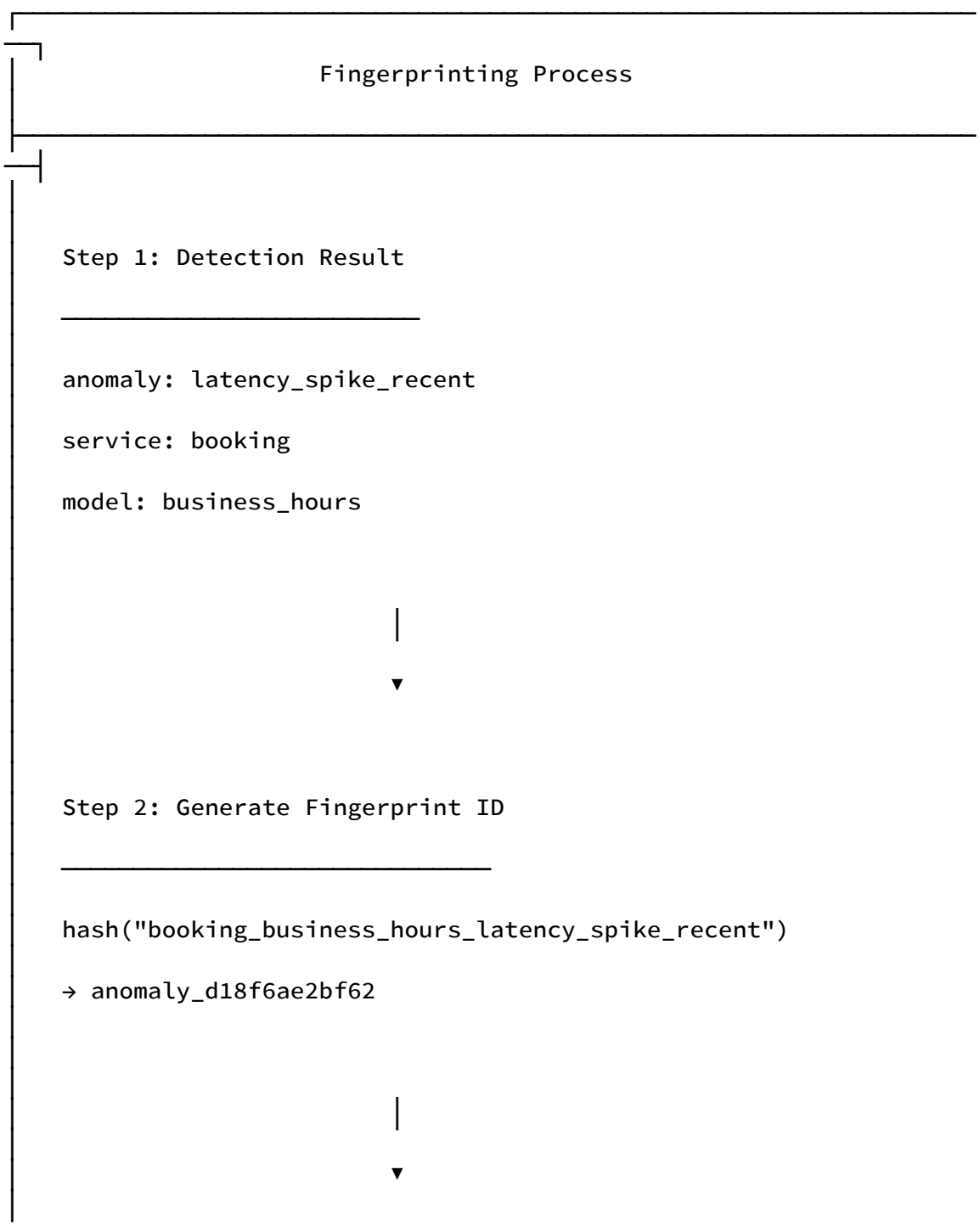
└──┘

What Each ID Enables

Capability	Fingerprint ID	Incident ID
Track same issue across time	✅	❌
Identify individual occurrences	❌	✅
Correlate related events	✅	❌
Calculate MTTR per incident	❌	✅
Find recurring patterns	✅	❌
Link alert to specific event	❌	✅

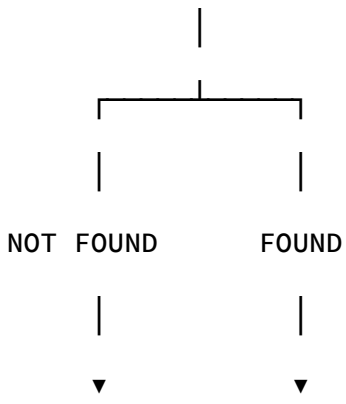
How Fingerprinting Works

The Complete Flow



Step 3: Database Lookup

```
SELECT * FROM anomaly_incidents
WHERE fingerprint_id = 'anomaly_d18f6ae2bf62'
  AND status IN ('SUSPECTED', 'OPEN', 'RECOVERING')
```



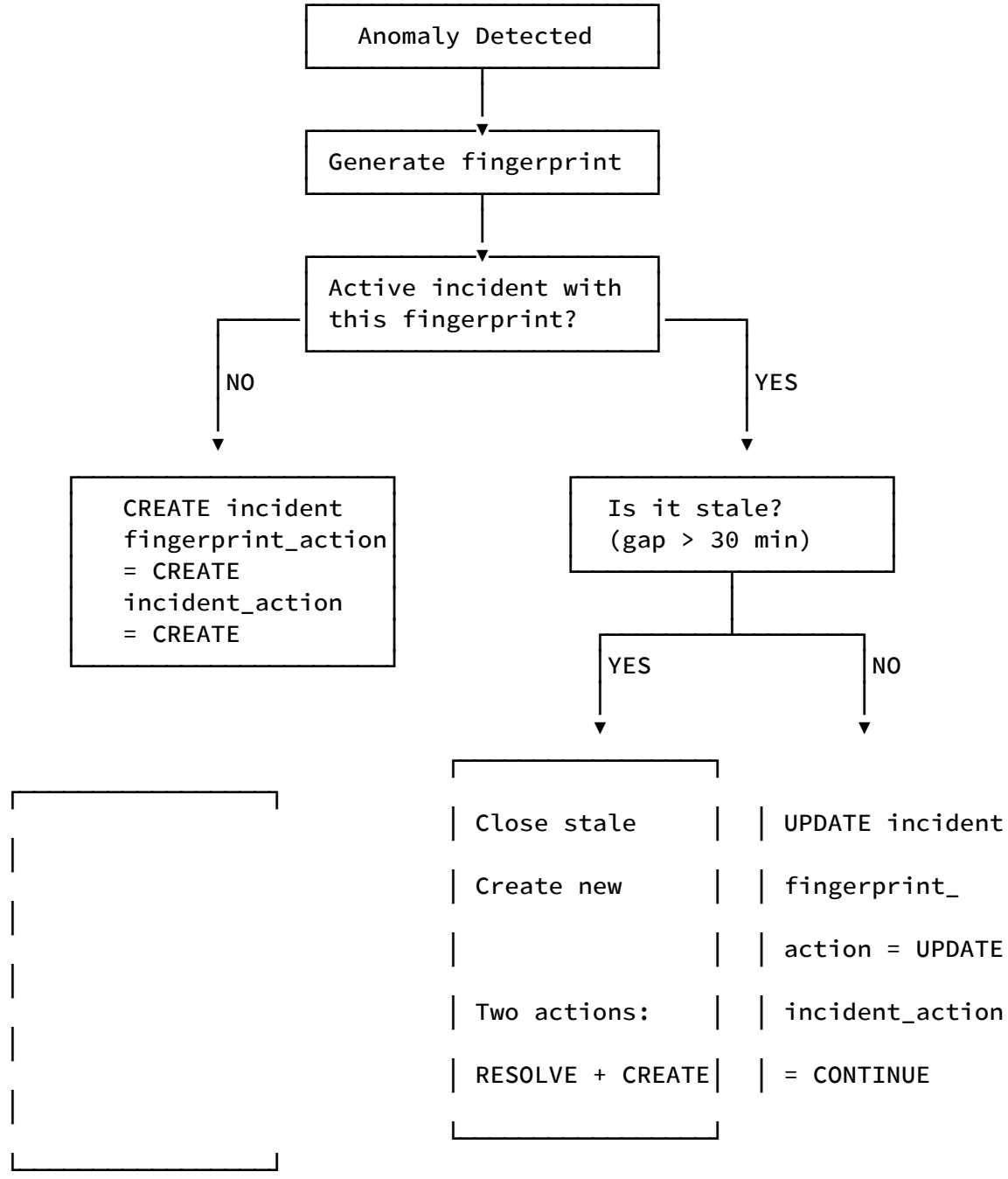
Step 4a: Create New

- Generate new incident_id
- Set status = SUSPECTED
- Initialize counters
- Set first_seen = now

Step 4b: Update Existing

- Keep same incident_id
- Check staleness
- Increment occurrence_count
- Update last_updated
- Handle state transitions

Decision Tree



Database Schema

The fingerprinting system uses SQLite for persistence:

```
CREATE TABLE anomaly_incidents (
    -- Identity
    fingerprint_id TEXT NOT NULL,          -- Pattern identifier
    incident_id TEXT PRIMARY KEY,          -- Occurrence identifier

    -- Context
    service_name TEXT NOT NULL,            -- Service (e.g.,
"booking")
    anomaly_name TEXT NOT NULL,            -- Pattern name
    detected_by_model TEXT,                -- Time period model

    -- State
    status TEXT NOT NULL,                  -- SUSPECTED, OPEN,
RECOVERING, CLOSED
    severity TEXT NOT NULL,                -- low, medium, high,
critical

    -- Timestamps
    first_seen TIMESTAMP NOT NULL,         -- When incident started
    last_updated TIMESTAMP NOT NULL,       -- Most recent detection
    resolved_at TIMESTAMP NULL,            -- When closed (null if
active)

    -- Counters
    occurrence_count INTEGER NOT NULL,     -- Total detections
    consecutive_detections INTEGER NOT NULL, -- In a row (for
confirmation)
    missed_cycles INTEGER NOT NULL,        -- Not detected (for
grace period)

    -- Optional data
    current_value REAL,                    -- Current metric value
    threshold_value REAL,                  -- Threshold if
applicable
    confidence_score REAL,                 -- ML confidence
    detection_method TEXT,                 -- How it was detected
    description TEXT,                      -- Human-readable
description
    metadata TEXT                          -- JSON for extra data
);

-- Indexes for fast lookups
CREATE INDEX idx_fingerprint_status ON
anomaly_incidents(fingerprint_id, status);
CREATE INDEX idx_service_timeline ON anomaly_incidents(service_name,
first_seen DESC);
CREATE INDEX idx_active_incidents ON anomaly_incidents(status,
```

```
last_updated DESC)
    WHERE status IN ('SUSPECTED', 'OPEN', 'RECOVERING');
```

Why SQLite?

Advantage	Description
Simple	No external dependencies
Reliable	ACID transactions built-in
Fast	Excellent for read-heavy workloads
Portable	Single file, easy to backup
Lightweight	Minimal resource usage

For most deployments, SQLite handles the expected volume easily (< 1000 active incidents).

Staleness Check

What Is Staleness?

When the time gap between detections exceeds a threshold, the system considers the incident “stale” - meaning it’s probably not the same ongoing issue, even if it’s the same pattern type.

Why Staleness Matters

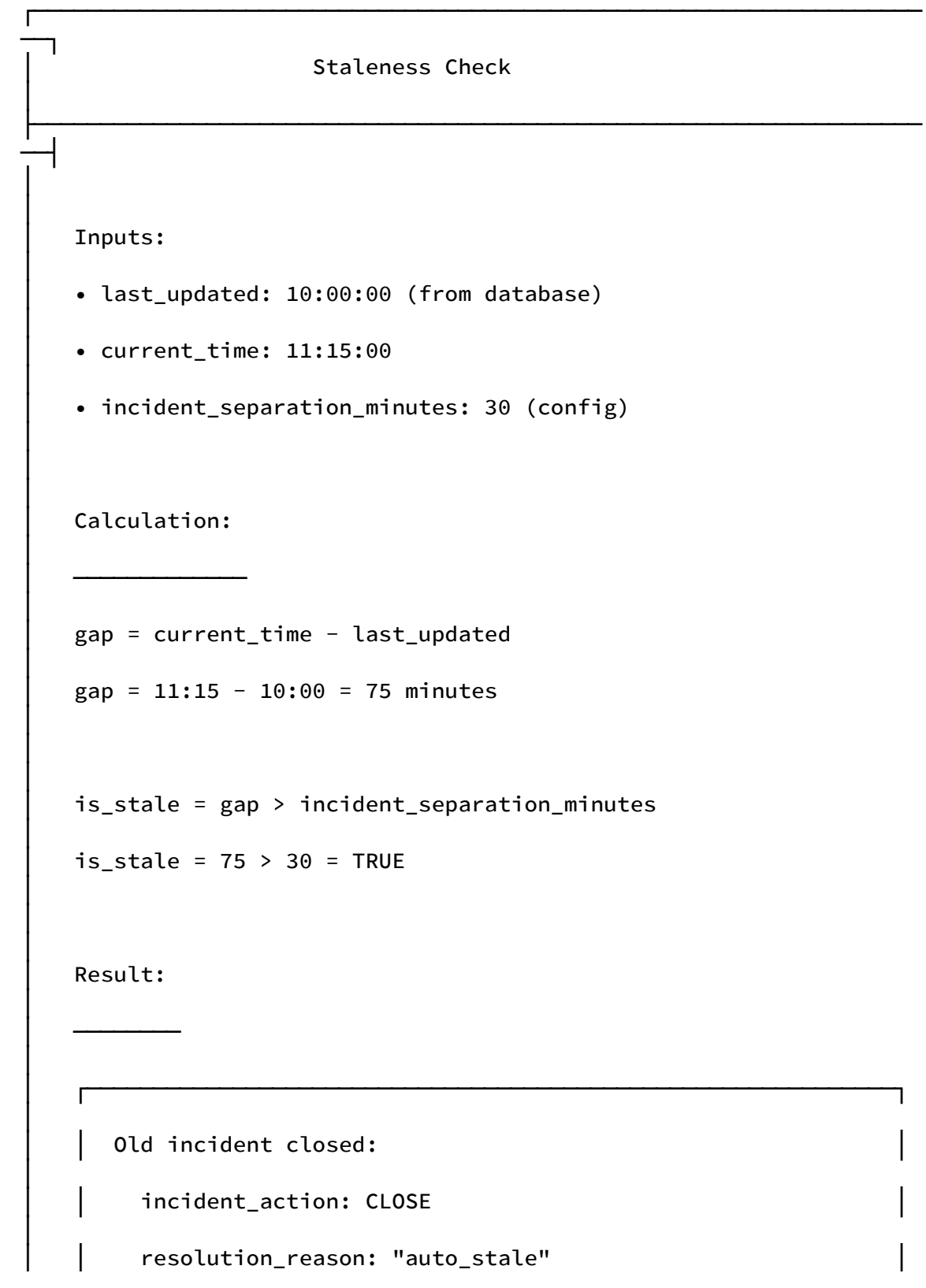
Without staleness check:

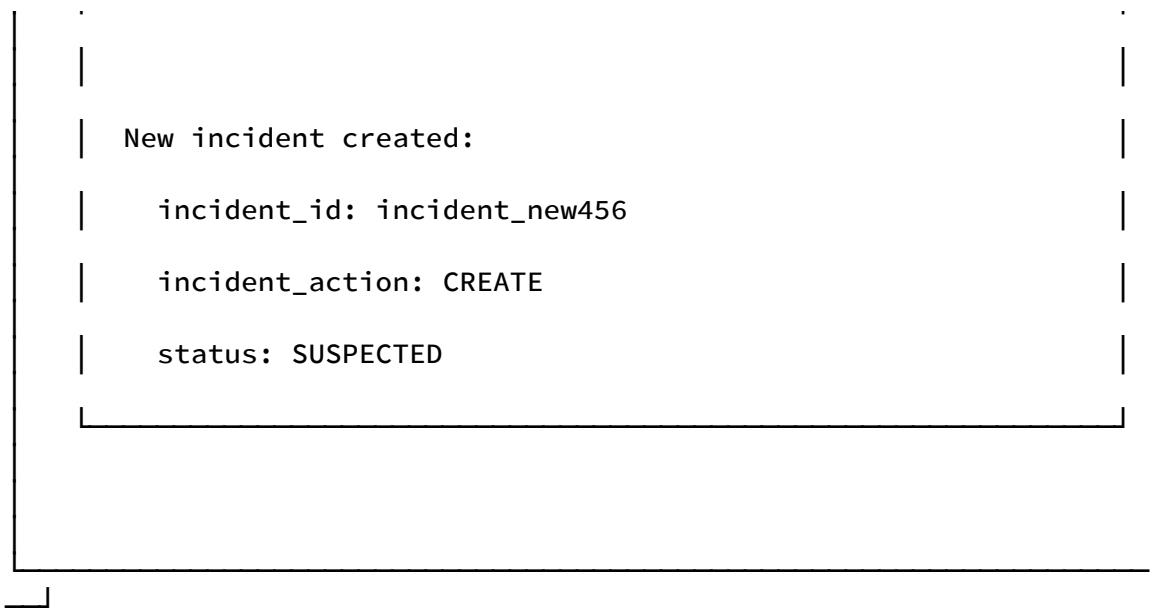
```
10:00  Latency spike detected → incident_123 created
10:03  Latency spike detected → incident_123 continues
10:06  Latency spike resolved

(long gap - different issue)

15:00  Latency spike detected → incident_123 continues??? NO!
       This is likely a different issue, not the same one from 5
hours ago!
```

With staleness check:

```
10:00  Latency spike detected → incident_123 created
10:03  Latency spike detected → incident_123 continues
10:06  Latency spike resolved

(long gap - different issue)

15:00  Latency spike detected
       Gap check: 15:00 - 10:06 = 4h 54min > 30min threshold
       Action: Close incident_123 as "auto_stale", create
incident_456
       Now we have two separate incidents (correct!)
```

Staleness Flow





Resolution Reasons

Reason	Description	When It Happens
`resolved`	Normal resolution	Anomaly cleared after grace period
`auto_stale`	Stale incident closed	Time gap exceeded threshold, pattern reappeared
`suspected_expired`	Never confirmed	SUSPECTED state expired without confirmation

Per-Anomaly Fingerprinting Fields

Each anomaly in the output includes comprehensive fingerprinting metadata:

```
{
  "anomalies": {
    "latency_spike_recent": {
      "type": "consolidated",
      "severity": "high",
      "description": "Latency spike: 450ms (normally 120ms)",

      "fingerprint_id": "anomaly_d18f6ae2bf62",
      "fingerprint_action": "UPDATE",

      "incident_id": "incident_31e9e23d4b2b",
      "incident_action": "CONTINUE",

      "status": "OPEN",
      "previous_status": "OPEN",

      "incident_duration_minutes": 15,
      "first_seen": "2025-12-17T13:45:00",
      "last_updated": "2025-12-17T14:00:00",

      "occurrence_count": 5,
      "consecutive_detections": 5,

      "is_confirmed": true,
      "newly_confirmed": false
    }
  }
}
```


Field Reference

Field	Type	Description
`fingerprint_id`	string	Pattern identifier (deterministic hash)
`fingerprint_action`	string	Pattern action: CREATE, UPDATE, RESOLVE
`incident_id`	string	Occurrence identifier (random UUID)
`incident_action`	string	Incident action: CREATE, CONTINUE, CLOSE

Field	Type	Description
`status`	string	Current status: SUSPECTED, OPEN, RECOVERING, CLOSED
`previous_status`	string	Status before this cycle
`incident_duration_minutes`	integer	Time since first_seen
`first_seen`	datetime	When this incident started
`last_updated`	datetime	Most recent detection time
`occurrence_count`	integer	Total times detected
`consecutive_detections`	integer	Detections in a row
`is_confirmed`	boolean	True if status is OPEN
`newly_confirmed`	boolean	True if just confirmed this cycle

Action Types

Fingerprint Actions

The fingerprint action describes what happened to the pattern tracking:

Action	Description	When
`CREATE`	New pattern encountered	First time this exact pattern is seen
`UPDATE`	Existing pattern updated	Pattern detected, already being tracked
`RESOLVE`	Pattern no longer active	Grace period exceeded, pattern cleared

Incident Actions

The incident action describes what happened to the specific occurrence:

Action	Description	When
`CREATE`	New incident created	New occurrence of a pattern (including after stale)
`CONTINUE`	Incident continues	Same occurrence still active
`CLOSE`	Incident closed	Occurrence resolved (grace period exceeded)

Combined Actions

The combination tells the full story:

fingerprint_action	incident_action	Meaning
CREATE	CREATE	Brand new pattern, new incident
UPDATE	CONTINUE	Ongoing issue, same incident
UPDATE	CREATE	Same pattern reappeared (was stale or resolved)
RESOLVE	CLOSE	Pattern cleared, incident resolved

Resolution Payload

When an incident is resolved, the system sends detailed resolution information:

```
{
  "fingerprinting": {
    "resolved_incidents": [
      {
        "fingerprint_id": "anomaly_d18f6ae2bf62",
        "incident_id": "incident_31e9e23d4b2b",
        "anomaly_name": "latency_spike_recent",

        "fingerprint_action": "RESOLVE",
        "incident_action": "CLOSE",

        "final_severity": "high",
        "resolved_at": "2025-12-17T14:30:00",
        "total_occurrences": 8,
        "incident_duration_minutes": 45,
        "first_seen": "2025-12-17T13:45:00",
        "service_name": "booking",
        "last_detected_by_model": "business_hours",
        "resolution_reason": "resolved"
      }
    ]
  }
}
```


Resolution Fields

Field	Type	Description
`fingerprint_id`	string	Pattern that was resolved
`incident_id`	string	Specific occurrence that closed
`anomaly_name`	string	Human-readable pattern name
`final_severity`	string	Severity at time of resolution
`resolved_at`	datetime	When the incident was closed
`total_occurrences`	integer	How many times detected during incident

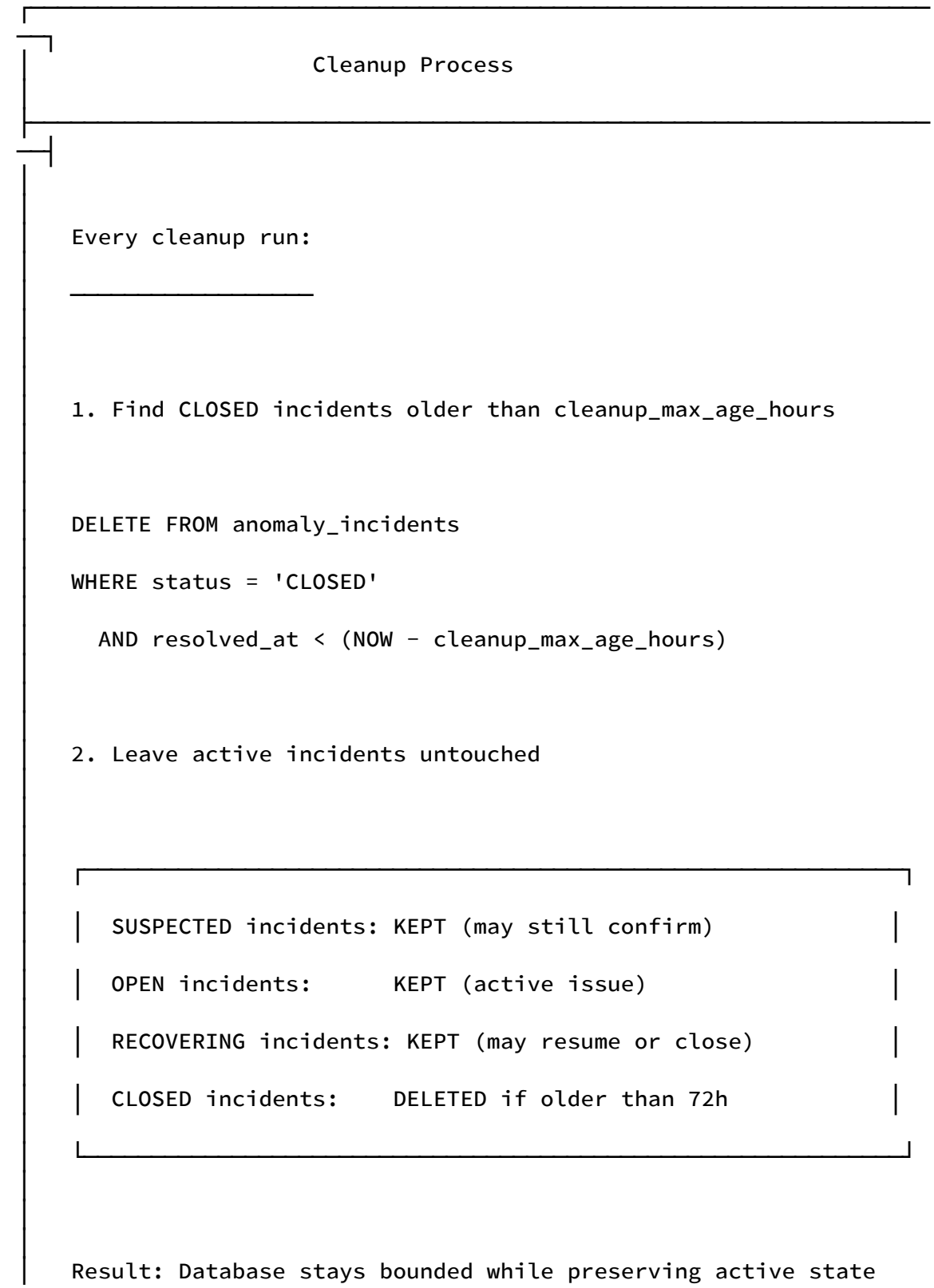
Field	Type	Description
`incident_duration_minutes`	integer	Total duration
`first_seen`	datetime	When it started
`service_name`	string	Affected service
`resolution_reason`	string	Why it was closed

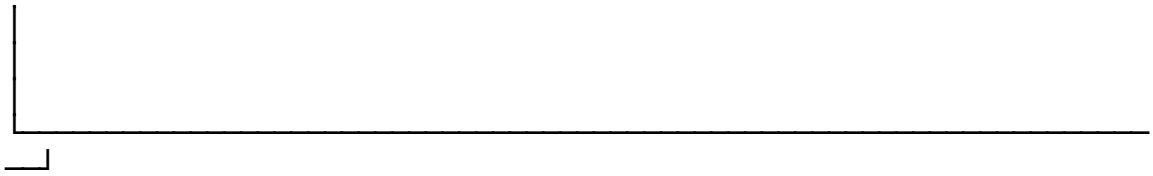
Database Cleanup

To prevent unbounded database growth, closed incidents are automatically cleaned up:

```
{
  "fingerprinting": {
    "cleanup_max_age_hours": 72
  }
}
```

Cleanup Process





Why 72 Hours?

Shorter (24h)	Longer (168h/1 week)
Less storage	More history
Faster queries	Slower queries
Less post-incident analysis	Better trend analysis
Risk of losing relevant history	More database bloat

72 hours (3 days) is a balance that:

- Covers most incident review periods
- Allows weekend incident analysis on Monday
- Keeps database size manageable

Configuration Reference

All fingerprinting settings in `config.json`:

```
{
  "fingerprinting": {
    "db_path": "./anomaly_state.db",
    "confirmation_cycles": 2,
    "resolution_grace_cycles": 3,
    "incident_separation_minutes": 30,
    "cleanup_max_age_hours": 72
  }
}
```

Setting	Default	Description
`db_path`	`./anomaly_state.db`	Path to SQLite database
`confirmation_cycles`	2	Detections required before alerting
`resolution_grace_cycles`	3	Non-detections before resolving
`incident_separation_minutes`	30	Gap threshold for staleness
`cleanup_max_age_hours`	72	Age threshold for cleanup

Summary

Fingerprinting provides the foundation for intelligent incident tracking:

1. **Fingerprint ID** enables pattern recognition across time
2. **Incident ID** enables precise occurrence tracking
3. **Staleness check** prevents confusion between separate issues
4. **Database persistence** maintains state across restarts
5. **Automatic cleanup** keeps the system healthy

Together, these mechanisms enable Yaga2 to:

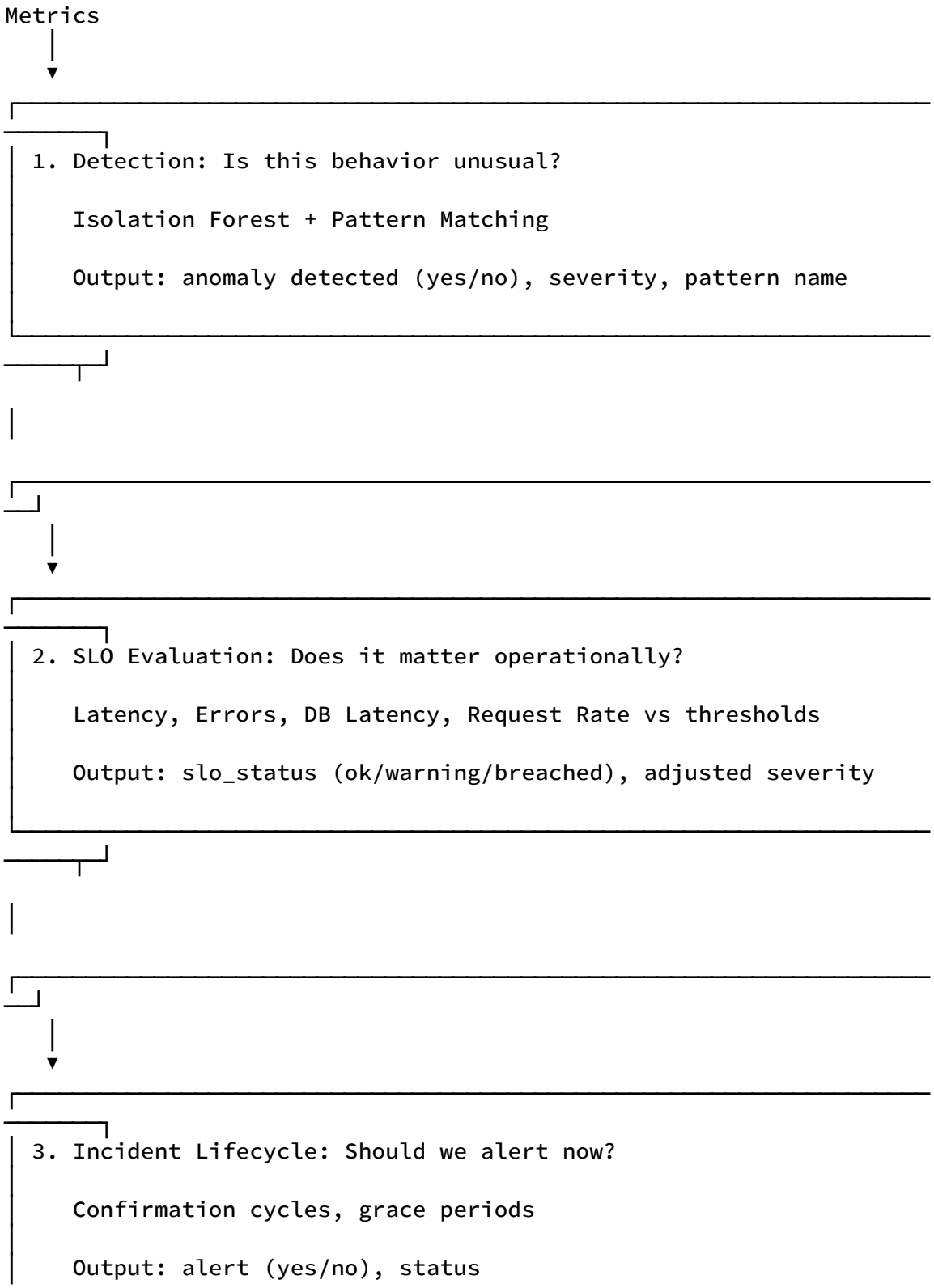
- Track recurring issues
- Calculate accurate metrics (MTTR, frequency)

- Prevent duplicate alerts
- Provide complete incident lifecycle visibility

Decision Matrix

Quick reference for how different conditions map to alert decisions.

End-to-End Decision Flow



```
(SUSPECTED/OPEN/RECOVERING/CLOSED)        │
└──────────────────────────────────────────────────────
──────┘
```

Detection Decision Matrix

Traffic	Latency	Errors	Pattern	Se
High	Normal	Normal	`traffic_surge_healthy`	lo
High	High	Normal	`traffic_surge_degrading`	hi
High	High	High	`traffic_surge_failing`	cr
Very Low	Any	Any	`traffic_cliff`	cr
Normal	High	Normal	`latency_spike_recent`	hi
Normal	High	Normal (deps healthy)	`internal_latency_issue`	hi
Normal	Normal	High	`error_rate_elevated`	hi
Normal	Normal	Very High	`error_rate_critical`	cr
Normal	Low	High	`fast_failure`	cr
Normal	Very Low	Very High	`fast_rejection`	cr
Normal	High (DB dominant)	Normal	`database_bottleneck`	hi
Normal	Normal (DB high)	Normal	`database_degradation`	m

SLO Severity Adjustment Matrix

ML Severity	Latency SLO	Error SLO	DB SLO	Final Severity
critical	ok	ok	ok	**low**
critical	ok	ok	warning	**low**
critical	warning	ok	ok	high
critical	ok	warning	ok	high
critical	breached	ok	ok	**critical**
critical	ok	breached	ok	**critical**
high	ok	ok	ok	**low**
high	warning	ok	ok	high
medium	ok	ok	ok	**low**
any	breached	breached	any	**critical**

Key Rule: SLO status `ok` → Final severity `low` (regardless of ML severity)

Incident State Decision Matrix

Current State	Anomaly Detected?	Consecutive	Action
None	Yes	1	CREATE (SUSPECTED)
SUSPECTED	Yes	< threshold	Stay SUSPECTED
SUSPECTED	Yes	≥ threshold	→ OPEN (alert)
SUSPECTED	No	< grace	Stay SUSPECTED
SUSPECTED	No	≥ grace	→ CLOSED (silent)
OPEN	Yes	any	Continue OPEN

Current State	Anomaly Detected?	Consecutive	Action
OPEN	No	1	→ RECOVERING
RECOVERING	Yes	any	→ OPEN (resume)
RECOVERING	No	< grace	Stay RECOVERING
RECOVERING	No	≥ grace	→ CLOSED (resolve)

Alert Decision Summary

Condition	Alert Sent?	Reason
First detection	No	SUSPECTED, waiting for confirmation
Second consecutive	Yes	Confirmed (OPEN)
Continuing OPEN	No	Already alerting
First non-detection	No	Grace period (RECOVERING)
Resolved	Resolution	CLOSED after grace period
Stale gap (>30 min)	New alert	Old closed, new SUSPECTED

Complete Example Scenarios

Scenario 1: Transient Spike (No Alert)

```
10:00 - Latency 450ms detected
        → ML: latency_spike_recent (high)
        → SLO: 450ms < 500ms acceptable → status: ok
        → Adjusted severity: low
        → Status: SUSPECTED (first detection)
        → Alert: ❌ NO

10:03 - Latency 120ms normal
        → No anomaly detected
        → Status: SUSPECTED (missed: 1)

10:06 - Latency 115ms normal
        → No anomaly detected
        → Status: SUSPECTED (missed: 2)

10:09 - Latency 118ms normal
        → Status: CLOSED (suspected_expired)
        → Alert: ❌ NO (never confirmed)
```

Scenario 2: Real Incident (Alert)

```
10:00 - Latency 850ms detected
        → ML: latency_spike_recent (critical)
        → SLO: 850ms > 800ms warning → status: warning
        → Adjusted severity: high
        → Status: SUSPECTED
        → Alert: ❌ NO (not confirmed)

10:03 - Latency 900ms detected
        → Status: OPEN (confirmed!)
        → Alert: ✅ YES

10:06 - Latency 920ms detected
        → Status: OPEN (continue)
        → Alert: Already sent

...

10:30 - Latency 200ms normal
        → Status: RECOVERING (grace: 1)

10:33 - Latency 180ms normal
        → Status: RECOVERING (grace: 2)

10:36 - Latency 175ms normal
        → Status: CLOSED (resolved)
        → Resolution: ✅ YES
```

Scenario 3: SLO Suppression (Low Priority)

```
10:00 - Latency 280ms detected
        → ML: latency_spike_recent (high)
        → SLO: 280ms < 300ms acceptable → status: ok
        → Adjusted severity: low (!!!)
        → Status: SUSPECTED
        → Alert: ❌ NO

10:03 - Latency 285ms detected
        → Status: OPEN (confirmed)
        → Severity: low
        → Alert: ✅ YES (low priority)
```

Quick Reference: When to Alert

Must be True	Description
Anomaly detected	ML flagged unusual behavior
Pattern matched	Known operational scenario
2+ consecutive	Confirmed, not transient
SLO evaluated	Operational impact assessed

Final Severity	Action
critical	PagerDuty, immediate action
high	Alert, investigate soon
low	Log only, informational
none	No action

Configuration Reference

Complete configuration reference for `config.json`.

Configuration File Location

The system searches for configuration in this order:

1. `CONFIG_FILE` environment variable
2. `./config.json` (current directory)
3. `./config/config.json`
4. `~/.smartbox/config.json`
5. `/etc/smartbox/config.json`

Core Sections

VictoriaMetrics

```
{
  "victoria_metrics": {
    "endpoint": "https://otel-metrics.production.smartbox.com",
    "timeout_seconds": 10,
    "max_retries": 3,
    "pool_connections": 20,
    "circuit_breaker_threshold": 5,
    "circuit_breaker_timeout_seconds": 300
  }
}
```

Field	Default	Description
`endpoint`	required	VictoriaMetrics server URL

Field	Default	Description
`timeout_seconds`	10	Request timeout
`max_retries`	3	Retry attempts
`circuit_breaker_threshold`	5	Failures before circuit opens

SLO Configuration

```
{
  "slos": {
    "enabled": true,
    "allow_downgrade_to_informational": true,
    "require_slo_breach_for_critical": true,
    "defaults": {
      "latency_acceptable_ms": 500,
      "latency_warning_ms": 800,
      "latency_critical_ms": 1000,
      "error_rate_acceptable": 0.005,
      "error_rate_warning": 0.01,
      "error_rate_critical": 0.02,
      "error_rate_floor": 0,
      "database_latency_floor_ms": 5.0,
      "database_latency_ratios": {
        "info": 1.5,
        "warning": 2.0,
        "high": 3.0,
        "critical": 5.0
      },
      "busy_period_factor": 1.5
    },
    "services": {
      "booking": {
        "latency_acceptable_ms": 300,
        "latency_critical_ms": 500,
        "error_rate_acceptable": 0.002,
        "error_rate_floor": 0.002
      }
    },
    "busy_periods": [
      {
        "start": "2024-12-20T00:00:00",
        "end": "2025-01-05T23:59:59"
      }
    ]
  }
}
```

Field	Default	Description
`enabled`	true	Enable SLO evaluation

Field	Default	Description
`allow_downgrade_to_informational`	true	Allow severity reduction when SLO ok
`require_slo_breach_for_critical`	true	Only critical if SLO breached
`latency_acceptable_ms`	500	Latency SLO acceptable threshold
`latency_critical_ms`	1000	Latency SLO critical threshold
`error_rate_acceptable`	0.005	Error rate acceptable (0.5%)
`error_rate_floor`	0	Error rate suppression floor
`database_latency_floor_ms`	5.0	DB latency noise floor

Fingerprinting

```
{
  "fingerprinting": {
    "db_path": "./anomaly_state.db",
    "cleanup_max_age_hours": 72,
    "incident_separation_minutes": 30,
    "confirmation_cycles": 2,
    "resolution_grace_cycles": 3
  }
}
```

Field	Default	Description
`db_path`	./anomaly_state.db	SQLite database path

Field	Default	Description
`cleanup_max_age_hours`	72	Hours before cleanup
`incident_separation_minutes`	30	Gap triggering new incident
`confirmation_cycles`	2	Cycles to confirm (send alert)
`resolution_grace_cycles`	3	Cycles before closing

Services

```
{
  "services": {
    "critical": ["booking", "search", "mobile-api"],
    "standard": ["friday", "gambit", "titan"],
    "micro": ["fa5"],
    "admin": ["m2-fr-adm", "m2-it-adm"],
    "core": ["m2-bb", "m2-fr"]
  }
}
```

Service categories affect default contamination rates:

Category	Contamination	Description
critical	0.03	Revenue-critical services
standard	0.05	Normal production
core	0.04	Platform infrastructure
admin	0.06	Administrative tools
micro	0.08	Low-traffic services

Dependencies

```
{
  "dependencies": {
    "graph": {
      "booking": ["search", "vms", "r2d2"],
      "vms": ["titan"],
      "search": ["catalog", "r2d2"]
    },
    "cascade_detection": {
      "enabled": true,
      "max_depth": 5
    }
  }
}
```

Model Configuration

```
{
  "model": {
    "models_directory": "./smartbox_models/",
    "min_training_samples": 500,
    "min_multivariate_samples": 1000,
    "default_contamination": 0.05,
    "default_n_estimators": 200,
    "contamination_by_service": {
      "booking": 0.02,
      "search": 0.04
    }
  }
}
```

Time Periods

```
{
  "time_periods": {
    "business_hours": {"start": 8, "end": 18, "weekdays_only":
true},
    "evening_hours": {"start": 18, "end": 22, "weekdays_only":
true},
    "night_hours": {"start": 22, "end": 6, "weekdays_only": true},
    "weekend_day": {"start": 8, "end": 22, "weekends_only": true},
    "weekend_night": {"start": 22, "end": 8, "weekends_only": true}
  }
}
```

Inference

```
{
  "inference": {
    "alerts_directory": "./alerts/",
    "max_workers": 3,
    "inter_service_delay_seconds": 0.2,
    "check_drift": false
  }
}
```

Environment Variables

Variable	Config Path	Description
`CONFIG_FILE`	-	Path to config file
`VM_ENDPOINT`	victoria_metrics.endpoint	VictoriaMetrics URL
`FINGERPRINT_DB`	fingerprinting.db_path	SQLite path
`OBSERVABILITY_URL`	observability_api.base_url	API server URL
`OBSERVABILITY_ENABLED`	observability_api.enabled	Enable API

Docker Configuration

```
environment:
  - TZ=UTC
  - CONFIG_PATH=/app/config.json
  - TRAIN_SCHEDULE=0 2 * * *
  - INFERENCE_SCHEDULE=*/10 * * * *
volumes:
  - ./smartbox_models:/app/smartbox_models
  - ./data:/app/data
  - ./config.json:/app/config.json
```

Adding New Services

1. Add to appropriate category in `services` section
2. Optionally add per-service SLO thresholds
3. Optionally add contamination override
4. Run training: `docker compose run --rm yaga train`
5. Verify: `docker compose run --rm yaga inference --verbose`

API Payload Reference

Reference for the JSON payload structure sent by the inference engine.

Alert Types

Type	Description
`anomaly_detected`	Anomalies detected for service
`no_anomaly`	Service is healthy
`metrics_unavailable`	Metrics collection failed

Top-Level Structure

```
{
  "alert_type": "anomaly_detected",
  "service_name": "booking",
  "timestamp": "2024-01-15T10:30:00",
  "time_period": "business_hours",
  "model_name": "business_hours",
  "model_type": "time_aware_5period",

  "anomaly_count": 1,
  "overall_severity": "high",

  "anomalies": { ... },
  "current_metrics": { ... },
  "slo_evaluation": { ... },
  "exception_context": { ... },
  "service_graph_context": { ... },
  "fingerprinting": { ... }
}
```

Current Metrics


```
{
  "current_metrics": {
    "request_rate": 52.7,
    "application_latency": 110.5,
    "dependency_latency": 1.4,
    "database_latency": 0.8,
    "error_rate": 0.0001
  }
}
```


Metric	Unit	Description
request_rate	req/s	Requests per second
application_latency	ms	Server processing time
dependency_latency	ms	External dependency latency
database_latency	ms	Database query time
error_rate	ratio	Error rate (0.05 = 5%)

Anomaly Object


```
{
  "latency_spike_recent": {
    "type": "consolidated",
    "root_metric": "application_latency",
    "severity": "high",
    "confidence": 0.80,
    "score": -0.5,
    "signal_count": 2,

    "description": "Latency degradation: 636ms (92nd percentile)",
    "interpretation": "Latency recently increased...",
    "pattern_name": "latency_spike_recent",

    "detection_signals": [ ... ],
    "recommended_actions": [ ... ],
    "comparison_data": { ... },

    "fingerprint_id": "anomaly_d18f6ae2bf62",
    "incident_id": "incident_31e9e23d4b2b",
    "status": "OPEN",
    "occurrence_count": 5,
    "is_confirmed": true
  }
}
```

Detection Signals


```
{
  "detection_signals": [
    {
      "method": "isolation_forest",
      "type": "ml_isolation",
      "severity": "low",
      "score": -0.01,
      "direction": "high",
      "percentile": 91.7
    },
    {
      "method": "named_pattern_matching",
      "type": "multivariate_pattern",
      "severity": "high",
      "score": -0.5,
      "pattern": "latency_spike_recent"
    }
  ]
}
```

SLO Evaluation


```
{
  "slo_evaluation": {
    "original_severity": "critical",
    "adjusted_severity": "low",
    "severity_changed": true,
    "slo_status": "ok",
    "slo_proximity": 0.56,
    "operational_impact": "informational",

    "latency_evaluation": {
      "status": "ok",
      "value": 280.0,
      "threshold_acceptable": 300,
      "proximity": 0.93
    },
    "error_rate_evaluation": {
      "status": "ok",
      "value": 0.001,
      "value_percent": "0.10%",
      "within_acceptable": true
    },
    "database_latency_evaluation": {
      "status": "warning",
      "value_ms": 25.0,
      "baseline_mean_ms": 10.0,
      "ratio": 2.5
    },
    "request_rate_evaluation": {
      "status": "ok",
      "type": "normal",
      "value_rps": 52.7,
      "baseline_mean_rps": 50.0,
      "ratio": 1.05
    },

    "explanation": "Severity adjusted from critical to low..."
  }
}
```

Cascade Analysis


```
{
  "cascade_analysis": {
    "is_cascade": true,
    "root_cause_service": "titan",
    "affected_chain": ["titan", "vms"],
    "cascade_type": "upstream_cascade",
    "confidence": 0.85,
    "propagation_path": [
      {"service": "titan", "has_anomaly": true, "anomaly_type":
"database_bottleneck"},
      {"service": "vms", "has_anomaly": true, "anomaly_type":
"downstream_cascade"}
    ]
  }
}
```

Fingerprinting


```
{
  "fingerprinting": {
    "service_name": "booking",
    "model_name": "business_hours",
    "timestamp": "2025-12-17T13:56:06",
    "overall_action": "UPDATE",
    "total_active_incidents": 1,
    "total_alerting_incidents": 1,

    "action_summary": {
      "incident_creates": 0,
      "incident_continues": 1,
      "incident_closes": 0,
      "newly_confirmed": 0
    },
    "status_summary": {
      "suspected": 0,
      "confirmed": 1,
      "recovering": 0
    },
    "resolved_incidents": [],
    "newly_confirmed_incidents": []
  }
}
```


Exception Context

Present when error SLO breached:

```
{
  "exception_context": {
    "service_name": "search",
    "timestamp": "2024-01-15T10:30:00",
    "total_exception_rate": 0.35,
    "exception_count": 3,
    "top_exceptions": [
      {
        "type": "Smartbox\\Search\\R2D2\\Exception\\R2D2Exception",
        "short_name": "R2D2Exception",
        "rate": 0.217,
        "percentage": 62.0
      }
    ],
    "query_successful": true
  }
}
```

Service Graph Context

Present when client latency SLO breached:

```
{
  "service_graph_context": {
    "service_name": "cmhub",
    "total_request_rate": 2.1,
    "routes": [
      {
        "server": "r2d2",
        "route": "roomavailabilitylistener",
        "request_rate": 0.117,
        "avg_latency_ms": 29.0,
        "percentage": 5.6
      }
    ],
    "top_route": { ... },
    "slowest_route": { ... },
    "summary": "Service graph for cmhub..."
  }
}
```

Resolution Payload


```
{
  "resolved_incidents": [
    {
      "fingerprint_id": "anomaly_061598e9ca91",
      "incident_id": "incident_abc123def456",
      "anomaly_name": "database_degradation",
      "fingerprint_action": "RESOLVE",
      "incident_action": "CLOSE",
      "final_severity": "medium",
      "resolved_at": "2025-12-17T14:30:00",
      "total_occurrences": 5,
      "incident_duration_minutes": 45,
      "first_seen": "2025-12-17T13:45:00",
      "resolution_reason": "resolved"
    }
  ]
}
```


Severity Values

Severity	Priority	Description
critical	1	Immediate action required
high	2	Investigate promptly
medium	3	Monitor closely
low	4	Informational
none	5	No anomaly

Named Patterns

Pattern	Severity	Trigger Condition
`traffic_surge_healthy`	low	High traffic, normal latency/errors

Pattern	Severity	Trigger Condition
`traffic_surge_degrading`	high	High traffic, high latency
`traffic_surge_failing`	critical	High traffic, high latency, high errors
`traffic_cliff`	critical	Very low traffic
`latency_spike_recent`	high	Normal traffic, high latency
`internal_latency_issue`	high	High latency, healthy deps
`error_rate_elevated`	high	Elevated error rate
`error_rate_critical`	critical	Very high error rate
`fast_failure`	critical	Low latency, high errors
`fast_rejection`	critical	Very low latency, very high errors
`database_bottleneck`	high	High DB latency, DB dominant
`database_degradation`	medium	High DB latency, compensating
`upstream_cascade`	high	High latency + upstream anomaly

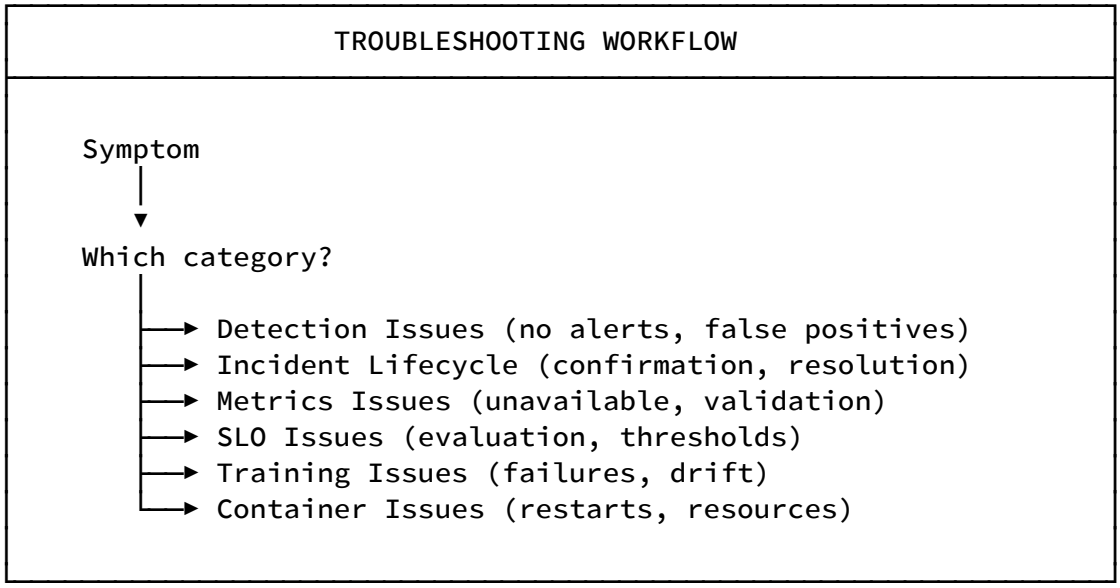
Troubleshooting

This guide helps you diagnose and resolve common issues with the anomaly detection system. Each section follows a systematic approach: identify symptoms, understand root causes, and apply targeted solutions.

How to Use This Guide

When troubleshooting:

1. **Identify the symptom** - What behavior are you observing?
2. **Gather evidence** - Collect relevant logs and configuration
3. **Understand the cause** - Why is this happening?
4. **Apply the fix** - Make targeted changes
5. **Verify the solution** - Confirm the issue is resolved



Detection Issues

No Anomalies Detected (When Expected)

Symptoms:

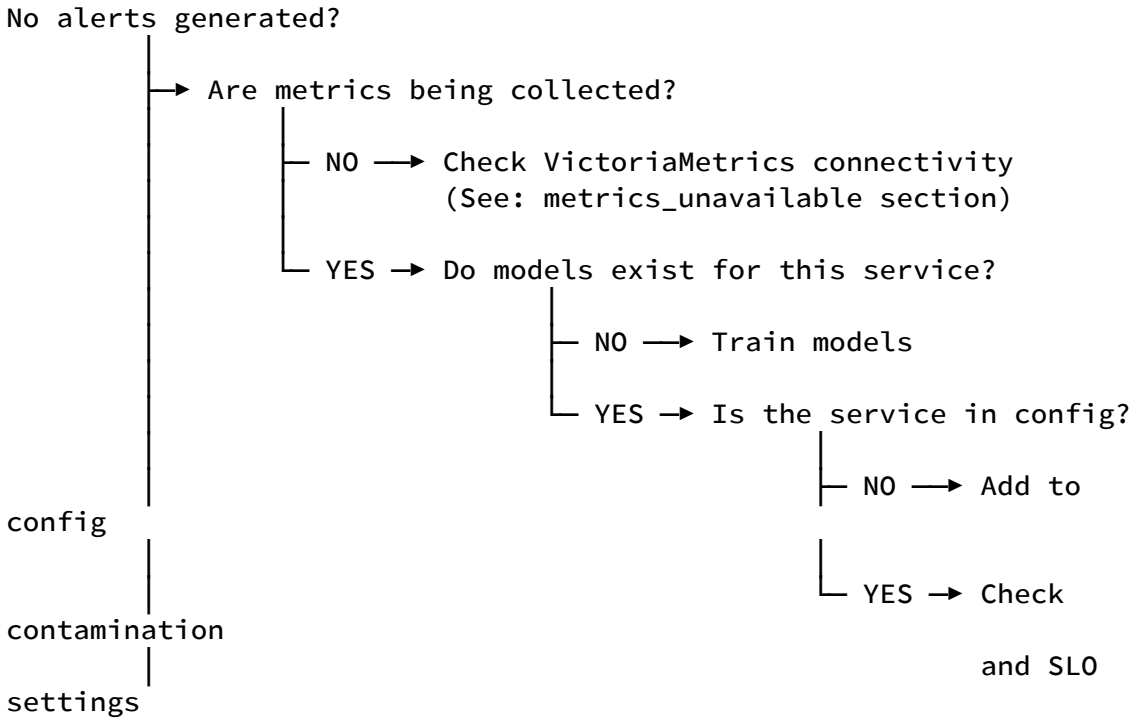
- Real incidents are occurring but the system generates no alerts
- Dashboards show problems but Yaga2 reports `alert_type: "no_anomaly"`
- Users report issues but your alerting is silent

Why This Happens:

The detection pipeline has multiple stages where anomalies can be "filtered out":

```
Metrics Collection → ML Detection → Pattern Matching → SLO
Evaluation → Alerting
         │                  │                │                │
         │                  │                │                │
     No data?          No model?       No pattern?      SLO
suppressed?
     (fail here)       (fail here)    (no match)       (low severity)
```

Diagnostic Decision Tree:



Step 1: Verify models exist:

```
# Check if models directory exists for your service
ls -la smartbox_models/<service-name>/

# Expected output shows time period subdirectories:
# drwxr-xr-x  business_hours/
# drwxr-xr-x  evening_hours/
# drwxr-xr-x  night_hours/
# drwxr-xr-x  weekend_day/
# drwxr-xr-x  weekend_night/
```

If no models exist, train them:

```
docker compose run --rm yaga train
```

Step 2: Check inference logs:

```
# View recent inference activity
docker compose exec yaga tail -50 /app/logs/inference.log

# Look for your service
docker compose exec yaga grep "<service-name>"
/app/logs/inference.log | tail -20
```

Step 3: Run verbose inference:

```
# This shows detailed detection output
docker compose run --rm yaga inference --verbose
```

Look for output like:

```
Service: booking
  Time period: business_hours
  Metrics collected: ✓
  ML detection: No anomalies (all scores normal)
  Pattern matching: No patterns matched
  Result: alert_type = no_anomaly
```

Common Causes and Solutions:

Cause	How to Identify	Solution
No trained model	`ls smartbox_models/<service>/` is empty	Run `docker compose run --rm yaga train`
Service not in config	Service not listed in `config.json`	Add to appropriate `services` category
Contamination too high	Model trained with high contamination (e.g., 0.10)	Lower to 0.03-0.05 and retrain
Model stale	Model trained on old data; current behavior differs	Retrain with recent data
Wrong time period	Checking business_hours during night	Verify timezone configuration

Cause	How to Identify	Solution
SLO suppression	Anomaly detected but SLO says “ok”	Check SLO thresholds (may be too lenient)

Example Investigation:

```
Scenario: booking service had a 5-minute outage but no alert was
generated

Step 1: Check models
$ ls smartbox_models/booking/
business_hours/  evening_hours/  night_hours/  weekend_day/
weekend_night/
✓ Models exist

Step 2: Check if service is in config
$ grep -A5 '"services"' config.json
"services": {
  "critical": ["booking", "search"],
  ...
}
✓ Service is configured

Step 3: Run verbose inference during the issue
$ docker compose run --rm yaga inference --verbose
...
booking: ML detected anomaly (score: -0.45, severity: high)
booking: SLO evaluation: latency 250ms < 500ms acceptable → status:
ok
booking: Severity adjusted: high → low (SLO status ok)
...

Diagnosis: SLO threshold too lenient (500ms acceptable)
Fix: Lower latency_acceptable_ms to 200ms
```

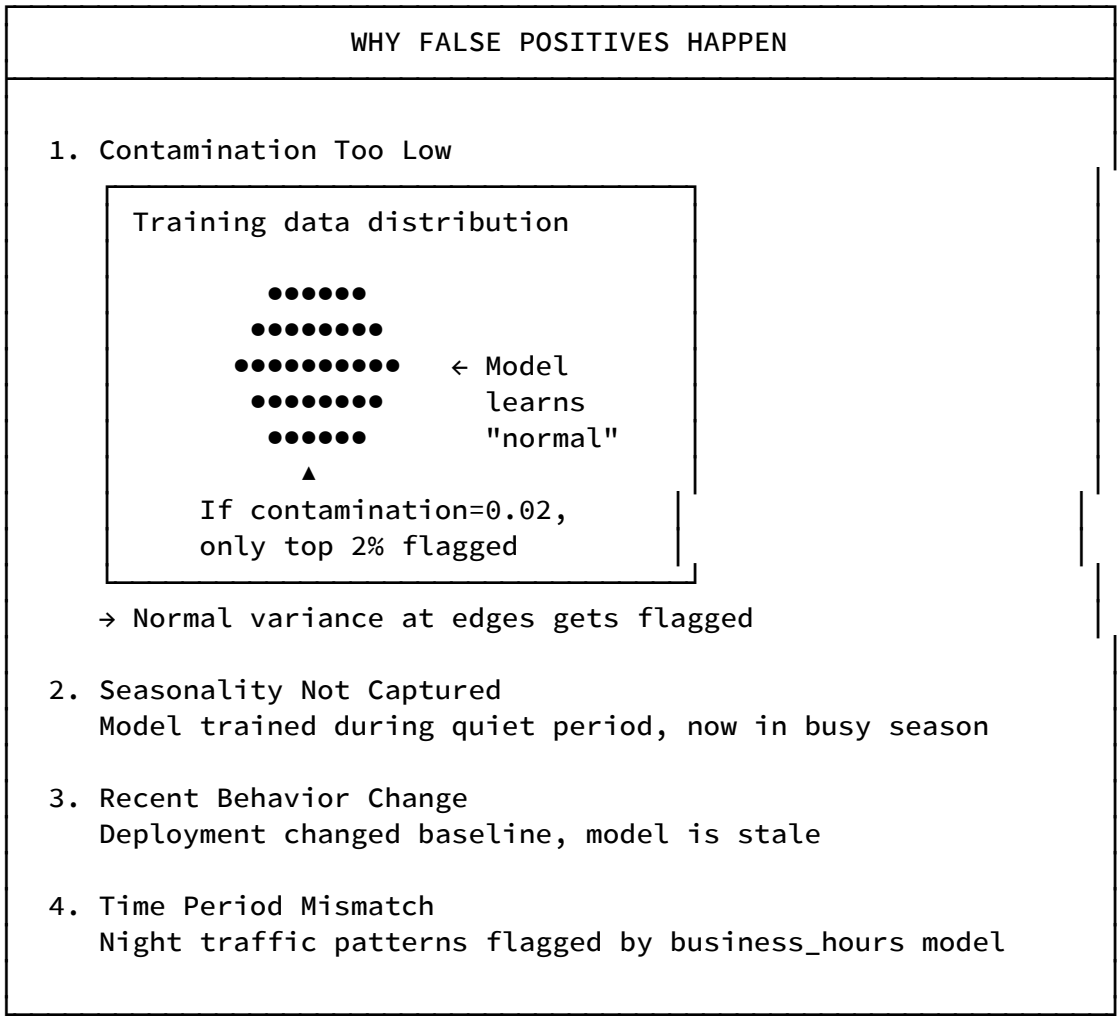
Too Many False Positives

Symptoms:

- Alerts fire during normal operation
- Team ignores alerts due to noise (alert fatigue)
- Anomalies detected don't correlate with real problems

Understanding False Positives:

False positives occur when the ML model flags behavior as "anomalous" when it's actually normal. This happens because:



Solution 1: Increase contamination rate

The contamination rate tells the model what percentage of training data to consider "anomalous". Higher values = less sensitive detection.

```
{
  "model": {
    "contamination_by_service": {
      "noisy-service": 0.08
    }
  }
}
```

Contamination	Sensitivity	Best For
0.02-0.03	Very high	Critical services where every alert matters
0.05	Balanced	Most production services
0.08-0.10	Lower	Noisy services, variable traffic

Solution 2: Adjust SLO thresholds

SLO evaluation can suppress anomalies that don't have operational impact:

```
{
  "slos": {
    "services": {
      "noisy-service": {
        "latency_acceptable_ms": 600,
        "error_rate_floor": 0.005
      }
    }
  }
}
```

This tells the system: "Even if ML detects an anomaly, don't alert unless latency exceeds 600ms or error rate exceeds 0.5%."

Solution 3: Retrain models

After changing configuration, always retrain:

```
docker compose run --rm yaga train
```

Measuring Improvement:

Track your false positive rate before and after changes:

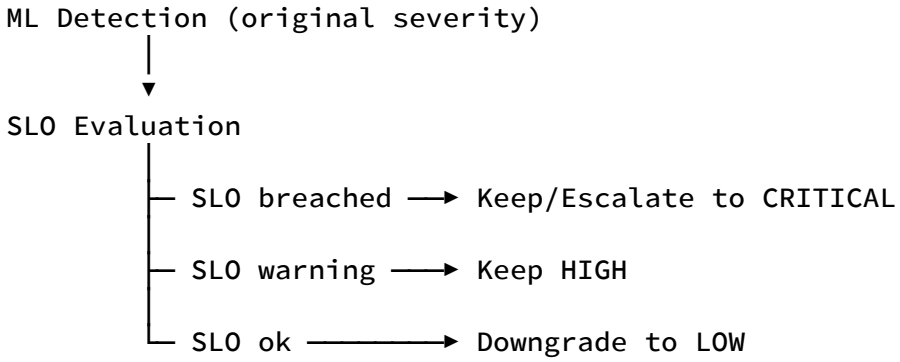
```
Before tuning:
  Total alerts (7 days): 145
  Actionable: 12 (8%)
  False positives: 133 (92%)

After increasing contamination 0.03 → 0.06:
  Total alerts (7 days): 34
  Actionable: 11 (32%)
  False positives: 23 (68%)

After adjusting SLO thresholds:
  Total alerts (7 days): 18
  Actionable: 10 (56%)
  False positives: 8 (44%)
```

Alerts Always Low Severity

Symptom:

- All alerts show `severity: low`
- Never see `high` or `critical` alerts
- Real incidents don't escalate properly

Understanding Severity Flow:



When SLO status is `ok` (all metrics within acceptable thresholds), severity is **always** downgraded to `low`. This is intentional—if metrics are operationally

acceptable, the alert shouldn't be high priority.

The Problem:

If your SLO thresholds are too lenient, anomalies will never breach them:

```
Example: Latency threshold too high

Current latency: 450ms (genuinely slow for this service)
SLO acceptable: 500ms (too lenient)
SLO status: "ok" (450 < 500)
Result: severity = low (even though users are impacted)
```

Solution: Tighten SLO thresholds

Review and lower your acceptable thresholds:

```
{
  "slos": {
    "services": {
      "booking": {
        "latency_acceptable_ms": 200,
        "latency_warning_ms": 300,
        "latency_critical_ms": 400,
        "error_rate_acceptable": 0.002,
        "error_rate_warning": 0.005,
        "error_rate_critical": 0.01
      }
    }
  }
}
```

Before vs After:

```
BEFORE (lenient thresholds):
  Latency acceptable: 500ms
  Current latency: 350ms
  SLO status: ok (350 < 500)
  ML severity: high → Adjusted: low

AFTER (tightened thresholds):
  Latency acceptable: 200ms
  Current latency: 350ms
  SLO status: warning (350 > 200, < 400)
  ML severity: high → Adjusted: high (kept)
```

Finding the Right Thresholds:

1. Look at your historical latency distribution:

   ```
   # In VictoriaMetrics, query p50, p90, p99 for your service
   ```

2. Set thresholds based on percentiles:
   - `acceptable` : Around p75 (allows normal variance)
   - `warning` : Around p90 (getting concerning)
   - `critical` : Around p99 (definitely a problem)

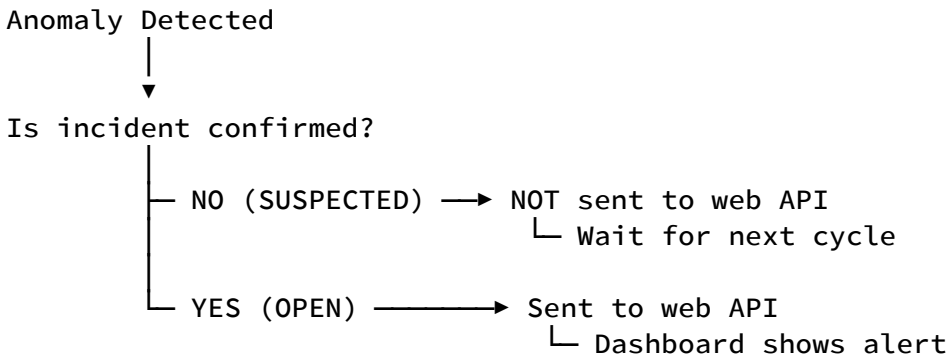
Incident Lifecycle Issues

Alerts Not Being Sent

Symptom:

- Verbose output shows anomaly detected
- Web API never receives the alert
- Dashboard shows no incidents

Understanding Alert Flow:



Diagnostic Steps:

Step 1: Check if anomaly is in SUSPECTED state

```
docker compose run --rm yaga inference --verbose
```

Look for:

```
fingerprint_action: CREATE
status: SUSPECTED
is_confirmed: false
cycles_to_confirm: 1
```

If `status: SUSPECTED`, the anomaly hasn't been confirmed yet. Wait for the next detection cycle.

Step 2: Check web API connectivity

```
# Test if web API is reachable
docker compose exec yaga curl http://observability-api:8000/health

# Expected: {"status": "healthy"}
```

If this fails, check:

- Is the observability API running?
- Is the network configured correctly?
- Are there firewall rules blocking traffic?

Step 3: Check if API is enabled

```
{
  "observability_api": {
    "enabled": true,
    "base_url": "http://observability-api:8000"
  }
}
```

If `enabled: false`, alerts won't be sent.

Step 4: Check for API errors in logs

```
docker compose exec yaga grep -i "api" /app/logs/inference.log |
tail -20

# Look for:
# - "API call failed"
# - "Connection refused"
# - "Timeout"
```

Orphaned Incidents in Web API

Symptom:

- Web API shows OPEN incidents that never resolve
- Incidents stuck in dashboard for days/weeks
- `consecutive_detections = 1` on stuck incidents

Why This Happens:

Before v1.3.2, SUSPECTED incidents were sent to the web API. If they expired without confirmation, no resolution was sent:

```
OLD BEHAVIOR (pre-v1.3.2):
──────────────────────────
10:00  Anomaly detected → SUSPECTED incident created
10:00  Alert sent to web API → Web API creates OPEN incident
10:03  Anomaly not detected → SUSPECTED expires
10:03  No resolution sent → Web API incident stays OPEN forever!

NEW BEHAVIOR (v1.3.2+):
──────────────────────────
10:00  Anomaly detected → SUSPECTED incident created
       (NOT sent to web API - waiting for confirmation)
10:03  Anomaly not detected → SUSPECTED expires silently
       Web API never knew about it → No orphan created
```

Identifying Orphaned Incidents:

Query your web API database:

```
-- Find orphaned incidents (never confirmed but sent)
SELECT incident_id, service_name, created_at, consecutive_detections
FROM incidents
WHERE status = 'OPEN'
  AND consecutive_detections = 1
  AND created_at < NOW() - INTERVAL '24 hours';
```

Solutions:

1. **Manual cleanup**: Close orphaned incidents in the web API
2. **Wait for upgrade**: v1.3.2+ won't create new orphans
3. **Automated cleanup**: Add a job to close incidents older than X days with `consecutive_detections = 1`

Incidents Restarting Unexpectedly

Symptom:

- Same anomaly pattern creates multiple incident IDs
- Incident history shows many short incidents instead of one long one
- Resolution reason shows `auto_stale`

Understanding Staleness:

The system considers an incident "stale" if the time gap since last detection exceeds `incident_separation_minutes` (default: 30):

```
Timeline with 45-minute gap:
──────────────────────────────────────────────────────────────

10:00  Anomaly detected → incident_abc created (SUSPECTED)
10:03  Detected again → incident_abc confirmed (OPEN)
10:06  Detected again → incident_abc continues
...
10:30  Last detection
       (service recovers but then has a second issue)
11:15  Anomaly detected → 45 min gap > 30 min threshold
                        → incident_abc auto-closed (stale)
                        → incident_xyz created (new incident)
```

When This Is Correct vs. Problematic:

```
CORRECT (two separate issues):
──────────────────────────────

10:00-10:30: Database slow due to query
11:15-12:00: Database slow due to disk I/O

These ARE two different incidents - staleness helps track them
separately.


PROBLEMATIC (one issue, intermittent symptoms):
────────────────────────────────────────────────

10:00-10:30: Network flaky (detected)
10:30-11:15: Network stable (not detected)
11:15-11:45: Network flaky again (same issue)

This is ONE issue but creates multiple incidents.
```

Solution: Increase separation threshold

```
{
  "fingerprinting": {
    "incident_separation_minutes": 60
  }
}
```

Threshold	Use Case
15 min	Fast-resolving issues, want precise tracking
30 min	Default, good balance
60 min	Intermittent issues, prefer fewer incidents
120 min	Very intermittent, consolidate aggressively

Metrics Issues

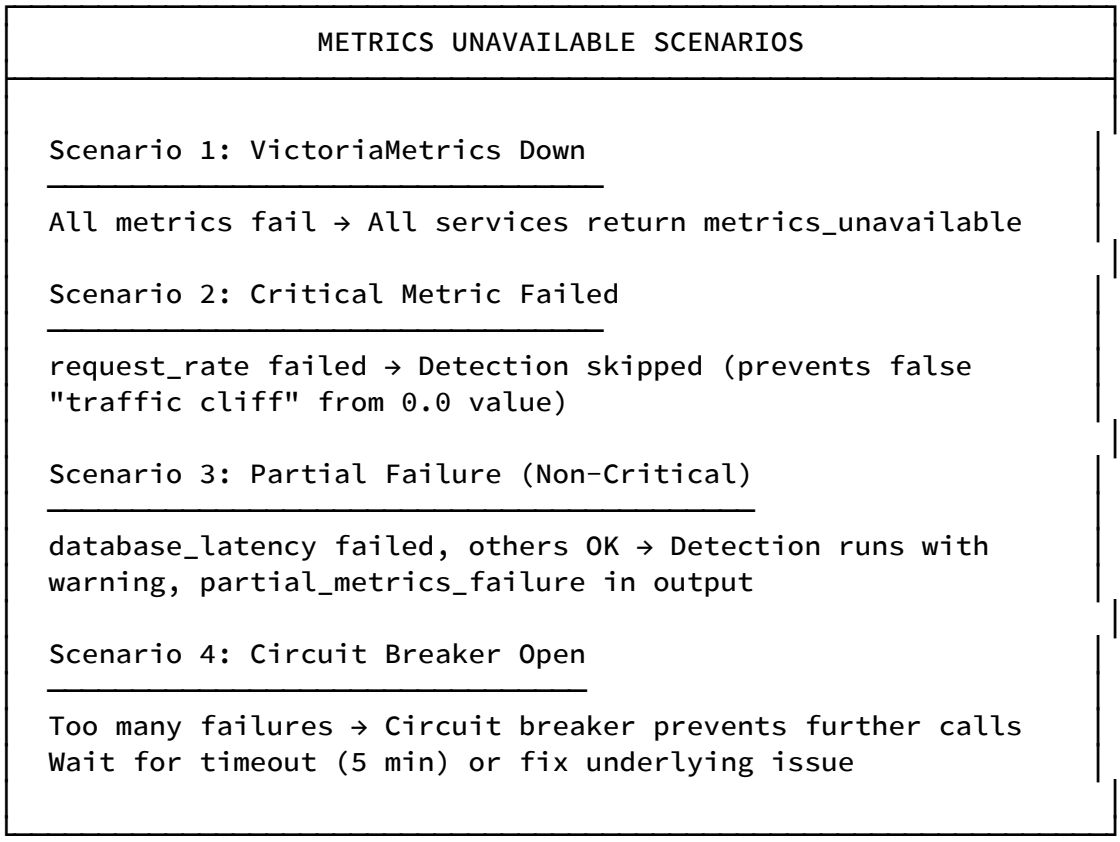
metrics_unavailable Alerts

Symptom:

- `alert_type: "metrics_unavailable"` instead of detection results
- Detection skipped for some/all services
- Circuit breaker messages in logs

Understanding This Alert:

The system returns `metrics_unavailable` when it cannot collect metrics reliably. This prevents false alerts—if we can't read metrics, we shouldn't guess.



Diagnostic Steps:

Step 1: Test VictoriaMetrics connectivity

```
# From inside the container
docker compose exec yaga curl -s
"http://vm:8428/api/v1/status/buildinfo"

# Expected: JSON with version info
# If this fails, VM is unreachable
```

Step 2: Check for circuit breaker

```
docker compose exec yaga tail -50 /app/logs/inference.log | grep -i
"circuit"

# Look for:
# "Circuit breaker OPEN - too many failures"
# "Circuit breaker will reset in X seconds"
```

Step 3: Check which metrics failed

Look at the output:

```
{
  "alert_type": "metrics_unavailable",
  "failed_metrics": ["request_rate", "application_latency"],
  "collection_errors": {
    "request_rate": "Connection timeout after 10s",
    "application_latency": "Connection timeout after 10s"
  }
}
```

Solutions:

Problem	Solution
VM down	Restart VictoriaMetrics, check VM health
Network issue	Check firewall, DNS, routing
Circuit breaker open	Wait 5 minutes or fix underlying issue
Timeout	Increase `timeout_seconds` in config

Circuit Breaker Behavior:

```
Normal Operation:
  Requests succeed → Circuit CLOSED

5 consecutive failures:
  Circuit OPENS → All requests fail fast (no retry)

After 5 minutes (configurable):
  Circuit HALF-OPEN → One request allowed

  If succeeds → Circuit CLOSES
  If fails → Circuit stays OPEN for another timeout
```

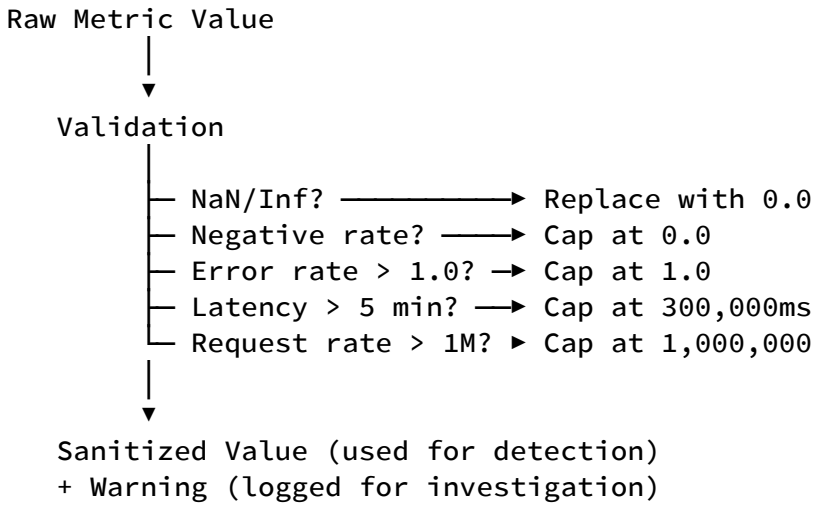
Validation Warnings

Symptom:

- `validation_warnings` array in output
- Messages like “capping at 1.0” or “using 0.0”

Understanding Validation:

Before metrics are processed, they’re validated at the inference boundary:



Example Warnings:

```
{
  "validation_warnings": [
    "error_rate: value 1.5 > 1.0, capping at 1.0",
    "application_latency: negative value -50, using 0.0",
    "request_rate: NaN detected, using 0.0"
  ]
}
```

Why This Happens:

Warning	Likely Cause	Investigation
Error rate > 1.0	Metric miscalculation upstream	Check error rate query definition
Negative latency	Clock skew or calculation bug	Check latency metric source
NaN/Inf	Division by zero, no data	Check if service was down

Warning	Likely Cause	Investigation
Extreme values	Metric spike or bug	Verify in VictoriaMetrics UI

Action Required:

Validation warnings are **informational**. Detection still runs with sanitized values. However, frequent warnings indicate upstream data quality issues that should be investigated.

```
# Track warning frequency
docker compose exec yaga grep "validation_warnings"
/app/logs/inference.log | wc -l
```

SLO Issues

SLO Evaluation Failed

Symptom:

- Warning in logs about SLO evaluation
- Anomalies not being severity-adjusted
- Missing `slo_evaluation` in output

Common Causes:

1. SLO not enabled

```
{
  "slos": {
    "enabled": false
  }
}
```

Set to `true` to enable SLO evaluation.

2. Service not configured

If a service isn't in the SLO config, it uses defaults:

```
{
  "slos": {
    "services": {
      "booking": {
        "latency_acceptable_ms": 300
      }
    }
  }
}
```

Services not listed use values from `slos.defaults`.

3. Missing training statistics

SLO evaluation for database latency requires training baseline. If the model doesn't have statistics, evaluation is skipped.

```
# Check if training statistics exist
docker compose exec yaga cat
smartbox_models/<service>/business_hours/metadata.json | python -c
"import sys,json; d=json.load(sys.stdin); print(d.get('statistics',
{}))"
```

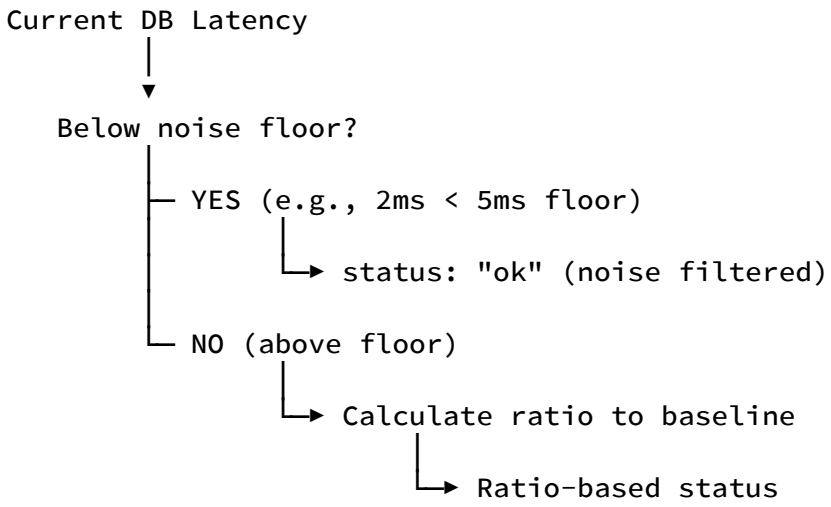
Database Latency Always OK

Symptom:

- ML detects database latency anomaly
- But `database_latency_evaluation.status: "ok"`
- Even when database is clearly slow

Understanding the Noise Floor:

Database latency uses a hybrid approach:



Why This Design:

For services with very fast databases (sub-millisecond), small absolute changes are operationally meaningless:

```
Without noise floor:
  Baseline: 0.3ms
  Current: 0.6ms
  Ratio: 2.0x → status: "warning"

  But 0.6ms is still FAST! This alert is noise.

With noise floor (5ms):
  Current: 0.6ms < 5ms floor
  status: "ok" (filtered)

  Operationally correct - 0.6ms is fine.
```

Adjusting the Floor:

For services with fast databases where you want to detect small changes:

```
{
  "slos": {
    "services": {
      "fast-db-service": {
        "database_latency_floor_ms": 1.0
      }
    }
  }
}
```

For services with slow databases where 5ms is normal:

```
{
  "slos": {
    "services": {
      "slow-db-service": {
        "database_latency_floor_ms": 10.0
      }
    }
  }
}
```

Checking Current Configuration:

```
{
  "database_latency_evaluation": {
    "value_ms": 2.0,
    "baseline_mean_ms": 1.5,
    "floor_ms": 5.0,
    "below_floor": true,
    "ratio": 0.0,
    "status": "ok"
  }
}
```

If `below_floor: true` , the value is being filtered. Lower the floor if needed.

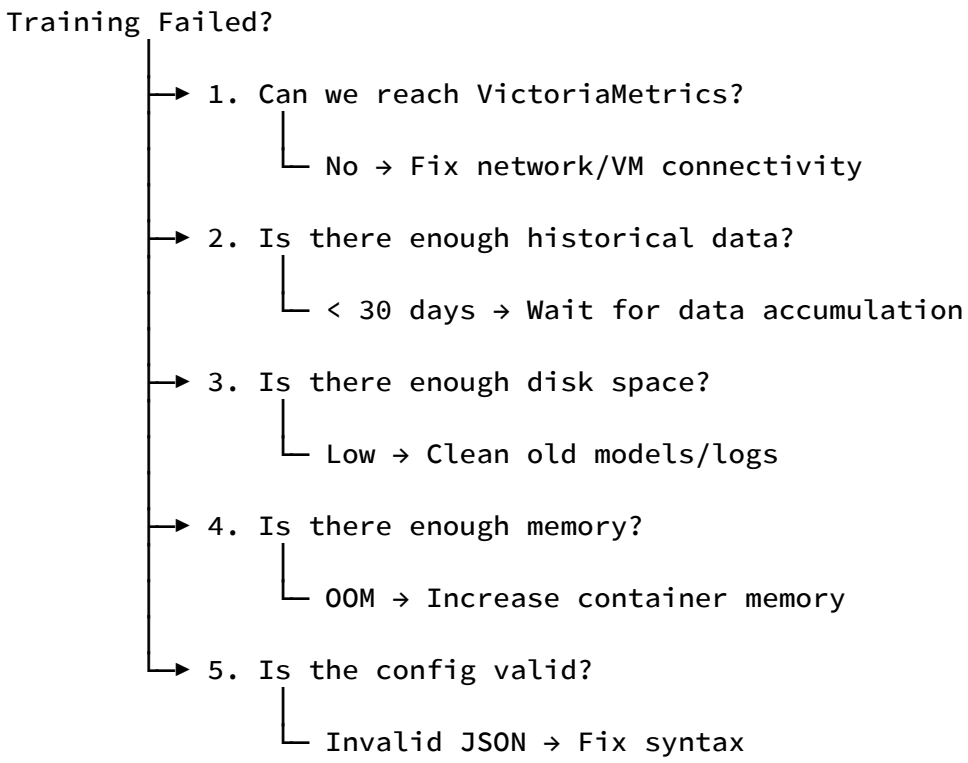
Training Issues

Training Fails

Symptoms:

- Models not updating (old timestamps)
- Training command exits with error
- No model files created

Diagnostic Checklist:



Step 1: Check training logs

```
docker compose exec yaga cat /app/logs/train.log

# Look for:
# - "Training failed for service X"
# - "Not enough data points"
# - "Connection refused"
# - "MemoryError"
```

Step 2: Check VictoriaMetrics

```
docker compose exec yaga curl -s
"http://vm:8428/api/v1/status/buildinfo"

# If this fails, VM is unreachable
```

Step 3: Check disk space

```
df -h

# Models typically need 10-50MB per service
# Ensure at least 1GB free
```

Step 4: Run training manually

```
docker compose run --rm yaga train 2>&1 | tee train_output.log

# This shows real-time output for diagnosis
```

Common Causes and Solutions:

Cause	Symptoms	Solution
VM unreachable	"Connection refused"	Check network, VM status
Not enough data	"Insufficient samples"	Wait for 30 days of data
Disk full	"No space left"	Clean old models: `rm -rf smartbox_models/old-service/`

Cause	Symptoms	Solution
Memory exhausted	“MemoryError” or OOM killed	Increase memory limit in docker-compose
Invalid config	“JSON decode error”	Validate config.json syntax

Minimum Data Requirements:

Model Type	Minimum Samples	Typical Duration
Univariate (per-metric)	500	~2 days at 5-min intervals
Multivariate (combined)	1000	~4 days at 5-min intervals
Full time-period coverage	8640	30 days (captures weekly patterns)

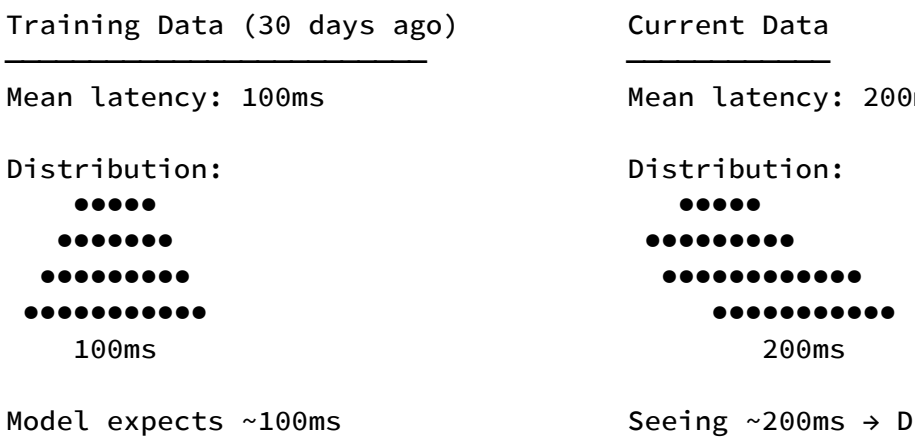
Model Drift Detected

Symptom:

- `drift_warning` in inference output
- Confidence scores reduced
- Recommendation to retrain

Understanding Drift:

Model drift occurs when production data differs significantly from training data:



Drift Score Interpretation:

Score	Status	Action
< 3	Normal	No action needed
3-5	Moderate drift	Monitor, consider retraining
> 5	Severe drift	Retrain soon

Output Example:

```
{
  "drift_warning": {
    "type": "model_drift",
    "overall_drift_score": 4.2,
    "affected_metrics": ["application_latency", "request_rate"],
    "recommendation": "WARNING: Moderate drift detected. Consider
retraining.",
    "confidence_penalty_applied": 0.15
  }
}
```

Why Drift Happens:

1. **Seasonal changes**: Holiday traffic patterns vs. normal
2. **Deployments**: New code changed latency characteristics
3. **Infrastructure**: Migrated to faster/slower hardware
4. **Business changes**: New features changed usage patterns

Solution: Retrain

```
docker compose run --rm yaga train
```

This replaces old models with ones trained on recent data.

Preventing Drift Issues:

- Schedule regular retraining (daily or weekly)
- Retrain after significant deployments
- Monitor drift scores in dashboards

Container Issues

Container Keeps Restarting

Symptoms:

- `docker compose ps` shows restart count increasing
- Container never stays "Up" for long
- Application logs show repeated startup/shutdown

Diagnostic Steps:

Step 1: Check exit code

```
docker compose ps -a

# Look for:
# smartbox-anomaly  Exit 1   (error exit)
# smartbox-anomaly  Exit 137 (OOM killed)
# smartbox-anomaly  Exit 0   (clean exit, shouldn't restart)
```

Step 2: Check recent logs

```
docker compose logs --tail 100

# Look for error messages near the end
```

Step 3: Check container events

```
docker events --filter container=smartbox-anomaly --since 1h
```

Common Exit Codes:

Exit Code	Meaning	Solution
0	Clean exit	Check restart policy, should be `unless-stopped`
1	Application error	Check logs for error message
137	OOM killed	Increase memory limit
139	Segmentation fault	Report bug, check for corruption

Solution by Cause:

Config file missing:

```
# Check if config exists
ls -la config.json

# If missing, create from template
cp config.example.json config.json
```

Invalid JSON in config:

```
# Validate JSON syntax
python -m json.tool config.json

# If error, fix the syntax issue
```

OOM killed:

```
# In docker-compose.yml, increase memory:
deploy:
  resources:
    limits:
      memory: 4G
```

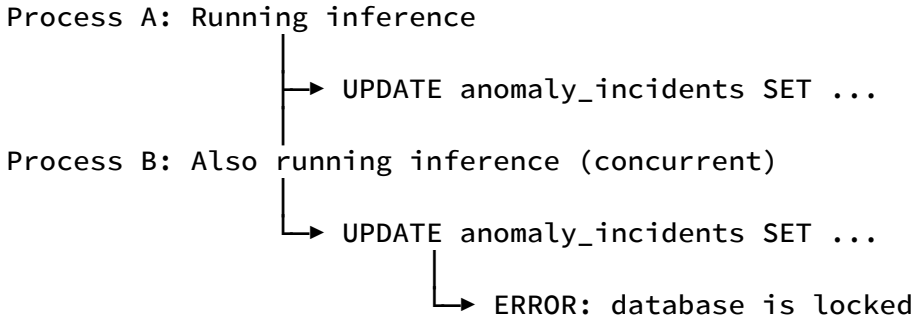
Database Locked

Symptom:

- `database is locked` error in logs
- SQLite error messages
- Fingerprinting failures

Understanding the Issue:

SQLite doesn't handle concurrent writes well. This happens when:



How This Can Happen:

1. **Manual inference + scheduled inference**: Running `docker compose run --rm yaga inference` while the scheduled inference is also running
2. **Multiple containers**: Two containers sharing the same volume
3. **Long-running queries**: One process holds lock while another waits

Solutions:

1. Only run one inference at a time:

```
# Check if inference is already running
docker compose ps | grep yaga

# If running, wait for it to finish
```

2. Use proper container orchestration:

Ensure only one inference container runs at a time via scheduling:

```
# In docker-compose.yml
command: ["scheduler"]  # Uses cron, prevents overlap
```

3. Check for stuck processes:

```
# Look for zombie inference processes
docker compose exec yaga ps aux | grep python

# If stuck, restart the container
docker compose restart
```

Corrupt Database Recovery:

If the database becomes corrupted:

```
# Backup current state
cp data/anomaly_state.db data/anomaly_state.db.backup

# Remove corrupted database
rm data/anomaly_state.db

# Restart - new database will be created
docker compose restart
```

Note: This loses incident history. All incidents will start fresh (new incident IDs).

Debugging

Enable Verbose Logging

For detailed diagnostic output:

Option 1: Command-line flag

```
docker compose run --rm yaga inference --verbose
```

This shows:

- Metrics being collected
- ML detection results
- Pattern matching output
- SLO evaluation details
- Fingerprinting actions

Option 2: Config file

```
{
  "logging": {
    "level": "DEBUG"
  }
}
```

Then rebuild and restart:

```
docker compose build
docker compose up -d
```

Log Level Guide:

Level	Use Case
INFO	Normal operation (default)
DEBUG	Troubleshooting, detailed output
WARNING	Only problems and warnings

Level	Use Case
ERROR	Only errors

Inspect Model Details

View model metadata:

```
# Check training info
docker compose exec yaga cat
/app/smartbox_models/<service>/business_hours/metadata.json | python
-m json.tool
```

Expected output:

```
{
  "service_name": "booking",
  "period": "business_hours",
  "trained_at": "2024-01-15T02:00:00Z",
  "training_samples": 8640,
  "contamination": 0.02,
  "n_estimators": 250,
  "statistics": {
    "application_latency": {
      "mean": 110.3,
      "std": 45.2,
      "p50": 105.0,
      "p95": 180.0,
      "p99": 250.0
    }
  },
  "calibrated_thresholds": {
    "critical": -0.58,
    "high": -0.32,
    "medium": -0.15,
    "low": -0.08
  }
}
```

Check model age:

```
# Find oldest and newest models
find smartbox_models -name "metadata.json" -exec sh -c 'echo {} &&
grep trained_at {}' \; | sort
```

Check Fingerprint Database

List active incidents:

```
docker compose exec yaga sqlite3 /app/data/anomaly_state.db \
  "SELECT fingerprint_id, status, severity, occurrence_count,
          datetime(first_seen), datetime(last_updated)
   FROM anomaly_incidents
   WHERE status != 'CLOSED'
   ORDER BY last_updated DESC;"
```

Count incidents by status:

```
docker compose exec yaga sqlite3 /app/data/anomaly_state.db \
  "SELECT status, COUNT(*) FROM anomaly_incidents GROUP BY status;"
```

Find long-running incidents:

```
docker compose exec yaga sqlite3 /app/data/anomaly_state.db \
  "SELECT fingerprint_id, service_name,
          ROUND((julianday('now') - julianday(first_seen)) * 24 *
60) as duration_minutes
   FROM anomaly_incidents
   WHERE status = 'OPEN'
   ORDER BY duration_minutes DESC
   LIMIT 10;"
```

Database schema reference:

```
-- Main table: anomaly_incidents
fingerprint_id TEXT       -- Pattern identifier (hash)
incident_id TEXT PRIMARY  -- Unique occurrence ID
service_name TEXT         -- Service name
anomaly_name TEXT         -- Anomaly type
status TEXT               -- SUSPECTED, OPEN, RECOVERING, CLOSED
severity TEXT             -- low, medium, high, critical
first_seen TIMESTAMP      -- When first detected
last_updated TIMESTAMP    -- Last detection time
resolved_at TIMESTAMP     -- When closed (NULL if open)
occurrence_count INTEGER  -- Times detected
consecutive_detections INTEGER -- For confirmation
missed_cycles INTEGER     -- For grace period
```

Quick Reference: Common Commands

```
# Health check
docker compose ps
docker compose exec yaga python -c "import smartbox_anomaly;
print('OK')"

# View logs
docker compose logs --tail 100
docker compose exec yaga tail -f /app/logs/inference.log

# Manual operations
docker compose run --rm yaga inference --verbose
docker compose run --rm yaga train

# Check models
ls -la smartbox_models/
docker compose exec yaga cat
smartbox_models/<service>/business_hours/metadata.json

# Check database
docker compose exec yaga sqlite3 /app/data/anomaly_state.db
".tables"
docker compose exec yaga sqlite3 /app/data/anomaly_state.db "SELECT
* FROM anomaly_incidents WHERE status='OPEN';"

# Network connectivity
docker compose exec yaga curl -s
http://vm:8428/api/v1/status/buildinfo
docker compose exec yaga curl http://observability-api:8000/health

# Container management
docker compose restart
docker compose build
docker compose up -d
```

Getting Help

If you can't resolve an issue:

1. **Collect diagnostic information**:

```
docker compose logs --tail 200 > diagnostic.log
docker compose run --rm yaga inference --verbose >>
diagnostic.log 2>&1
docker compose exec yaga cat config.json >> diagnostic.log
```

2. **Check documentation**:
    - Configuration Guide - SLO and threshold settings
    - Detection Pipeline - How detection works
    - Incident Lifecycle - State machine details
3. **File an issue** with:
    - Symptom description
    - Expected vs. actual behavior
    - Diagnostic logs
    - Configuration (sanitized)
    - Steps to reproduce