Unified Availability Model (UAM)

A Methodology for Calculating the Availability of Heterogeneous IT Systems

Revised and Extended Edition (v2.0, November 2025)

Alexey A. Nekludoff

AstraVerge Research

Email: an@astraverge.org

26 November 2025

Abstract

The Unified Availability Model (UAM) proposes a formal, extensible framework for evaluating the availability of heterogeneous information systems whose operational characteristics, performance indicators, and failure modes differ fundamentally across layers and functional domains. Unlike traditional SRE- and API-centric approaches that assume homogeneous metric spaces and rely primarily on latency–error abstractions, UAM introduces a generalizable normalization methodology capable of transforming diverse metric types into a unified, comparable scale.

The model is based on four foundational constructs — statistical baselines, SLA thresholds, physical system limits, and acceptable deviation ranges — each represented as a mathematically defined normalization operator. By combining these operators through a weighted linear composition, UAM enables consistent availability scoring for systems as diverse as REST/gRPC microservices, payment gateways, cryptographic HSM-signing modules, batch/ETL pipelines, ERP platforms (Oracle EBS, SAP), and EDI/IDoc integration channels.

At the higher level, UAM introduces the concept of a business-process contour — a structured composite of heterogeneous subsystems — and defines contour availability as a weighted aggregation of subsystem availability scores. This provides a unified and interpretable indicator of end-to-end business operability, compatible with real-world observability tools such as Prometheus, Zabbix, Grafana, and VictoriaMetrics.

The proposed model offers both theoretical novelty — by formalizing cross-system metric normalization — and practical applicability, providing engineering teams with a consistent methodology for measuring, comparing, and governing availability across complex

IT landscapes. UAM constitutes a step toward a generalized mathematical foundation for reliability assessment in multi-layered enterprise systems.

This document represents the revised and extended Version 2.0 of the Unified Availability Model (UAM), incorporating hierarchical coherence models and a formal treatment of fuzzy logic and neural-network limitations.

Contents

1 Introduction

Modern IT landscapes consist of **heterogeneous and multi-layered** systems, including (but not limited to) the following categories:

- API services and microservices,
- payment gateways,
- banking integrations,
- cryptographic and HSM services,
- batch processes and ETL pipelines,
- ERP systems (such as SAP S/4HANA, Oracle EBS),
- reporting and analytics systems.

Each of these categories has its own operation principles, workload patterns, and metric types. For this reason, **a single universal availability metric is not possible**.

However, it is possible to create **a unified methodological approach** in which:

- each system is evaluated using its own native metrics,
- these metrics are normalized to a common scale,
- the final availability score is calculated as a weighted sum.

This approach makes it possible to:

- fairly compare different types of systems,
- build an integrated availability score for an entire process contour,
- unify reporting and monitoring,
- easily extend the model to new classes of systems.

2 Key Definitions

2.1 System Availability

System availability A_i is a normalized score that represents the state of system i over a selected observation period. It takes a value from 0 to 1, or from 0% to 100%.

2.2 Metric

A **metric** M_{ij} is a quantitative indicator that describes the state of system i by parameter j.

2.3 Normalization

Normalization is the transformation of a raw metric M_{ij} into a normalized value $N_{ij} \in [0, 1]$ according to the rules defined for the specific system category.

2.4 Metric Weight

w_{ij} is a weight coefficient that defines the contribution of metric j to the final availability of system i, where:

$$\sum_j w_{ij} = 1.$$

2.5 Contour Availability

Contour availability A_{contour} is a weighted sum of the availability values of the systems that form a business-process chain.

3 Unified Availability Model: General Structure

Each system S_i has a set of metrics:

$$\{M_{i1}, M_{i2}, \ldots, M_{ik}\}.$$

Each metric is normalized according to:

$$N_{ij} = f_{norm}(M_{ij}),$$

where $N_{ij} \in [0, 1]$.

Each normalized metric has an assigned weight w_{ij}.

3.1 System Availability Formula

$$A_i = \sum_{j=1}^{k} w_{ij} \cdot N_{ij}.$$

The final value of A_i is expressed either in the range $[0, 1]$ or as a percentage.

3.2 Contour Availability Formula

Let the contour consist of n systems. Then:

$$A_{\text{contour}} = \sum_{i=1}^{n} W_i \cdot A_i,$$

where W_i is the weight of system i in the contour, and $\sum_i W_i = 1$.

4 Metric Normalization

In the Unified Availability Model (UAM), each metric M_{ij} is normalized to a value $N_{ij} \in [0, 1]$. The normalization is based on four key concepts:

1. stable baseline,
2. SLA thresholds,
3. physical limits,
4. acceptable deviations.

Below are the formal definitions and normalization rules that describe how raw metrics are transformed into normalized values.

4.1 Stable Baseline

Definition. A stable baseline for metric M is a statistically stable characteristic that reflects the typical system behavior under normal conditions, without anomalies.

The baseline shows the "normal" operation mode and is used as a reference point for detecting degradation. It is derived from historical data.

Normalization rules.

Let B be the baseline, calculated using one of the statistical methods:

- median: $B = \text{median}(M_{\text{hist}})$,
- 75th percentile: $B = \text{quantile}_{0.75}$,

- exponential smoothing: $B = \text{EWMA}(M_{\text{hist}})$,
- seasonal baselines (by hour, weekday, etc.).

Let k be a tolerance coefficient (usually 1.5–2). Then:

$$N_{\text{baseline}}(M) = \begin{cases} 1, & M \le kB, \\ \frac{kB}{M}, & M > kB. \end{cases} \tag{1}$$

4.2 SLA Thresholds

Definition. An SLA is a target value that defines the boundary between "acceptable" and "unacceptable" service quality.

Normalization rules.

Formally, SLA is expressed as:

$$M \le SLA \quad \text{(for latency)}, \qquad M \ge SLA \quad \text{(for success rate)}.$$

$$N_{\text{SLA}}(M) = \begin{cases} 1, & M \text{ does not violate SLA}, \\ 1 - \alpha(M - SLA), & M \text{ violates SLA}, \end{cases} \tag{2}$$

where α is a penalty coefficient.

4.3 Physical Limits

Definition. Physical limits are hardware or architectural constraints that do not depend on SLA.

Typical examples include:

- network bandwidth,
- maximum number of connections in a pool,
- CPU or IO limits,
- architectural RPS ceiling.

Normalization rules.

Let the physical range be:

$$M_{\text{min}}^{\text{phys}} \le M \le M_{\text{max}}^{\text{phys}}.$$

Then:

$$N_{\text{phys}}(M) = \frac{M_{\text{max}}^{\text{phys}} - M}{M_{\text{max}}^{\text{phys}} - M_{\text{min}}^{\text{phys}}}. \tag{3}$$

4.4 Acceptable Deviations

Definition. Acceptable deviations represent a range of values where small metric fluctuations are not considered degradation.

$$D_{\text{min}} \leq M \leq D_{\text{max}}.$$

Normalization rules.

$$N_{\text{dev}}(M) = \begin{cases} 1, & D_{\text{min}} \leq M \leq D_{\text{max}}, \\ 1 - \beta(M - D_{\text{max}}), & M > D_{\text{max}}, \\ 1 - \beta(D_{\text{min}} - M), & M < D_{\text{min}}, \end{cases} \tag{4}$$

where β is the penalty coefficient.

4.5 Final Normalization

For each metric, the final normalization is a weighted composition of the approaches:

$$N(M) = w_b N_{\text{baseline}}(M) + w_s N_{\text{SLA}}(M) + w_p N_{\text{phys}}(M) + w_d N_{\text{dev}}(M), \tag{5}$$

where $w_b + w_s + w_p + w_d = 1$.

5 Typical Normalization Strategies

5.1 Linear Normalization

$$N = 1 - \frac{M - M_{\text{min}}}{M_{\text{max}} - M_{\text{min}}}.$$

5.2 SLA-Based Normalization

$$N = \begin{cases} 1, & M \leq SLA, \\ 1 - k(M - SLA), & M > SLA. \end{cases}$$

5.3 Normalization with a Saturating Function

$$N = e^{-\alpha M}.$$

5.4 Example: Latency Normalization

As an example, consider the normalization of a latency metric.

$$N_{\text{latency}} = \begin{cases} 1, & L \le L_{\text{baseline}}, \\ \frac{L_{\max} - L}{L_{\max} - L_{\text{baseline}}}, & L > L_{\text{baseline}}. \end{cases}$$

6 Examples of Metric Normalization for Different System Types

The following examples show how normalization can be applied to four important classes of systems: API services, cryptographic services, batch processes, and ERP platforms such as Oracle EBS. These examples demonstrate how the concepts of baseline, SLA, physical limits, and acceptable deviations are used for real observability metrics.

6.1 API Services and Payment Gateways

For API services, the main metrics include:

- M_1: latency $p99$,
- M_2: success rate,
- M_3: share of 5xx errors,
- M_4: queue depth or backlog.

Baseline example:

$$B_{latency} = \text{median}(L_{\text{hist}}) = 85 \text{ ms}.$$

SLA example:

$$\text{latency}_{p99} \le 300 \text{ ms}.$$

Physical limit:

$$L_{\max}^{\text{phys}} = 2000 \text{ ms} \quad (\text{Nginx kernel timeout}).$$

Acceptable deviation:

$$D_{\max} = 2.0 \cdot B_{latency}.$$

Latency normalization:

$$N_{\text{API-lat}} = \begin{cases} 1, & L \le 2B_{latency}, \\ \frac{2B_{latency}}{L}, & L > 2B_{latency}. \end{cases}$$

6.2 Cryptographic Services (HSM, Signing)

Key metrics:

- M_1: HSM availability (online/offline),
- M_2: document signing time,
- M_3: signing error rate,
- M_4: key slot availability.

Baseline example:

$$B_{\text{sign}} = \text{median}(T_{\text{sign}}) = 42 \text{ ms}.$$

SLA:

$$T_{\text{sign}} \leq 150 \text{ ms}.$$

Physical limits:

$$T_{\text{min}}^{\text{phys}} = 15 \text{ ms}, \quad T_{\text{max}}^{\text{phys}} = 500 \text{ ms}.$$

Acceptable deviation:

$$D_{\text{max}} = 1.5 \cdot B_{\text{sign}}.$$

Normalization:

$$N_{\text{sign}} = \begin{cases} 1, & T \leq D_{\text{max}}, \\ 1 - 0.01(T - D_{\text{max}}), & T > D_{\text{max}}. \end{cases}$$

6.3 Batch Processes and Reporting

Key metrics:

- M_1: job execution time,
- M_2: completion within the time window,
- M_3: execution errors,
- M_4: backlog or queue size.

Baseline:

$$B_{\text{job}} = \text{median}(t_{\text{job,hist}}) = 17 \text{ min}.$$

SLA:

$$t_{\text{job}} \leq 30 \text{ min}.$$

Physical limit:

$$t_{\text{max}}^{\text{phys}} = 120 \text{ min} \quad \text{(batch window overflow)}.$$

Acceptable deviation:

$$D_{\text{max}} = 1.2 \cdot B_{\text{job}}.$$

Normalization:

$$N_{\text{batch}} = \begin{cases} 1, & t \leq D_{\text{max}}, \\ \frac{D_{\text{max}}}{t}, & t > D_{\text{max}}. \end{cases}$$

6.4 ERP / SAP S/4HANA

Key metrics:

- M_1: SAP Dialog Response Time (DB Time, CPU Time, Wait Time),
- M_2: Work Process Utilization (DIA, BTC, UPD, SPO),
- M_3: Queue Length (SM50/SM66),
- M_4: HANA DB latency and lock waits.

Baseline:

$$B_{\text{SAP}} = \text{median}(T_{\text{dialog}}) = 280 \text{ ms}.$$

SAP SLA:

$$T_{\text{dialog}} \leq 1000 \text{ ms}.$$

Physical limits:

$$T_{\text{min}}^{\text{phys}} = 50 \text{ ms}, \qquad T_{\text{max}}^{\text{phys}} = 5000 \text{ ms}.$$

Acceptable deviation:

$$D_{\text{max}} = 2.0 \cdot B_{\text{SAP}}.$$

Normalization:

$$N_{\text{SAP}} = \begin{cases} 1, & T \leq D_{\text{max}}, \\ 1 - 0.0005\,(T - D_{\text{max}}), & T > D_{\text{max}}. \end{cases}$$

6.5 ERP / Oracle EBS

Key metrics:

- M_1: database wait events (TX, TM, enq:...),
- M_2: active Concurrent Managers,
- M_3: workflow lag depth,

- M_4: session pool usage.

Baseline:

$$B_{\text{TX}} = \text{median}(\text{wait\_TX}) = 8 \text{ ms}.$$

SLA:

$$\text{wait\_TX} \leq 40 \text{ ms}.$$

Physical limit:

$$\text{wait\_TX}^{\text{max}} = 500 \text{ ms}.$$

Acceptable deviation:

$$D_{\text{max}} = 3 \cdot B_{\text{TX}}.$$

Normalization:

$$N_{\text{EBS}} = \begin{cases} 1, & W_{\text{TX}} \leq D_{\text{max}}, \\ 1 - 0.005(W_{\text{TX}} - D_{\text{max}}), & W_{\text{TX}} > D_{\text{max}}. \end{cases}$$

6.6 SAP IDoc / EDI Integrations

Key metrics:

- M_1: IDoc processing time (end-to-end latency),
- M_2: share of IDoc errors (status 51/68),
- M_3: inbound/outbound queue depth,
- M_4: lock conflicts during processing.

Baseline:

$$B_{\text{IDoc}} = \text{median}(T_{\text{idoc}}) = 1.8 \text{ s}.$$

SLA:

$$T_{\text{idoc}} \leq 5 \text{ s}.$$

Physical limits:

$$T_{\text{min}}^{\text{phys}} = 0.5 \text{ s}, \qquad T_{\text{max}}^{\text{phys}} = 30 \text{ s}.$$

Acceptable deviation:

$$D_{\text{max}} = 2.0 \cdot B_{\text{IDoc}}.$$

Normalization:

$$N_{\text{IDoc}} = \begin{cases} 1, & T \leq D_{\text{max}}, \\ 1 - 0.02\,(T - D_{\text{max}}), & T > D_{\text{max}}. \end{cases}$$

6.7 Integration with Elasticsearch / OpenSearch

Elasticsearch and OpenSearch can serve as complementary data sources for UAM because many systems produce structured operational logs rather than metrics. UAM relies on normalized numerical inputs, and search engines can supply these inputs in the form of aggregated event counts, latency distributions, or error classifications extracted from logs.

Why Elasticsearch/OpenSearch are useful for UAM

- **Structured log fields** allow deriving metrics not exposed by Prometheus (e.g., workflow errors, business failures, SAP IDoc status).
- **High-volume indexing** supports massive enterprise landscapes.
- **Powerful aggregations** (terms, date histogram, percentiles) allow reproducing UAM baselines and SLA distributions.
- **Long-term storage** is ideal for multi-month baseline calculation.

Example: Extracting error ratios

A Kibana/OpenSearch DSL query can compute error counts:

```
GET logs-*/_search
{
 "size": 0,
 "query": {
  "match": { "level": "ERROR" }
 },
 "aggs": {
  "errors": { "value_count": { "field": "message" } }
 }
}
```

Converted into a normalized metric:

$$N_{\text{err}} = \text{clamp}\left(1 - \frac{\text{errors}}{\text{threshold}}, 0, 1\right)$$

Example: Log-derived latency distribution

```
"aggs": {
 "latency_p95": {
  "percentiles": { "field": "latency_ms", "percents": [95] }
```

```
  }
 }
```

This can feed into UAM latency normalization:

$$N_{\text{lat}} = \frac{L_{\max} - L}{L_{\max} - B}$$

Baseline computation from logs

Elasticsearch/OpenSearch aggregations facilitate seasonal and multi-week baselines:

```
"aggs": {
 "daily": {
  "date_histogram": { "field": "@timestamp", "interval": "1d" },
  "aggs": {
   "median_latency": { "percentiles": { "field": "latency_ms",
     "percents": [50] } }
  }
 }
}
```

Exporting metrics to UAM

Three export mechanisms are typical:

- **Elastic → Prometheus exporter** (export aggregate values as Prometheus metrics)
- **Elastic → VictoriaMetrics via push** (send normalized metrics directly)
- **Prometheus → Elastic agent** (Elastic ingests Prometheus metrics + UAM metrics)

Once converted to metrics, Elasticsearch/OpenSearch derived values participate in the standard UAM formulas:

$$A_i = \sum_j w_{ij} N_{ij}$$

and

$$A_{\text{contour}} = \sum_i W_i A_i.$$

6.8 Integration with Kafka and RabbitMQ

Messaging platforms such as Kafka and RabbitMQ play a critical role in enterprise contours. Many subsystems (SAP, banking, CRM, payment gateways) depend on reliable message flow. UAM can directly incorporate queue metrics, lag metrics, and throughput metrics from these brokers.

Why messaging systems matter for UAM

- Message delays directly affect end-to-end contour availability.
- Queue backlogs indicate partial failures or slow consumers.
- Broker-level errors (rebalance storms, partition unavailability, consumer lag) become visible in normalized metrics.
- Kafka/RabbitMQ provide precise throughput/latency data.

Kafka

Kafka exposes rich metrics via:

- JMX exporters for Prometheus,
- Client-side metrics (producer/consumer),
- Broker/partition metrics.

Key Kafka metrics for UAM

- **Consumer Lag** — M_1
- **Message Throughput** — M_2
- **Rebalance Events** — M_3
- **Failed Produce/Consume Attempts** — M_4

Example normalization:

$$N_{\text{lag}} = \frac{D_{\max} - \text{lag}}{D_{\max}}$$

$$N_{\text{fail}} = 1 - \frac{\text{failures}}{\text{threshold}}$$

Kafka-derived availability

$$A_{\text{kafka}} = 0.40N_{\text{lag}} + 0.30N_{\text{throughput}} + 0.20N_{\text{fail}} + 0.10N_{\text{rebalance}}$$

This integrates seamlessly with contour-level availability.

RabbitMQ

RabbitMQ provides AMQP queue and channel metrics via Prometheus exporters or the built-in management API.

Key RabbitMQ metrics for UAM

- **Queue Depth** — M_1
- **Message Rate In/Out** — M_2
- **Unacked Messages** — M_3
- **Connection Errors** — M_4

Example normalization:

$$N_{\text{queue}} = \text{clamp}\left(1 - \frac{\text{queue depth}}{D_{\max}}, 0, 1\right)$$

$$N_{\text{unacked}} = \frac{U_{\max} - U}{U_{\max}}$$

RabbitMQ availability:

$$A_{\text{rabbit}} = 0.35N_{\text{queue}} + 0.25N_{\text{rate}} + 0.25N_{\text{unacked}} + 0.15N_{\text{conn}}$$

Message broker availability becomes one of the A_i values in:

$$A_{\text{contour}} = \sum_i W_i A_i.$$

Use cases in enterprise contours

- SAP IDoc → Kafka → microservices → ERP posting.
- Payment gateway → RabbitMQ → accounting.
- Workflow engines using Kafka for orchestration.

In all such cases, broker degradation directly impacts the business contour, which UAM captures via normalized queue, lag, and error metrics.

6.9 Comparison Table of Normalization Approaches

Parameter	**Baseline**	**SLA**	**Physical Limits / Deviations**
Source of value	Historical data	Contract / obligation	Architectural / hardware limits
Control type	Detecting degradation	Binary check of "normal"	Stability and resource limit control
Orientation	System behavior	Client expectations	Physical and operational limits
API example	p50/p75 latency baseline	$\text{latency}_{p99} \leq 300$ ms	Nginx timeout = 2000 ms
Batch example	median job time	job $\leq$ SLA window	max job = 120 min
EBS example	median TX wait	TX wait $\leq$ SLA	session pool = [0, 300]
Role in UAM	Basis for degradation detection	Strict penalties for violations	Soft penalty zone before failure

Table 1: Comparison of Normalization Approaches in UAM

7 System Types and Recommended Metrics

The tables below present recommended metrics and weight coefficients for different categories of information systems. These values are templates and may be adjusted for a specific IT landscape.

7.1 API Services and Payment Gateways

Metric	Symbol	Weight w_j
Success Rate	N_{succ}	0.40
Latency $p99$	N_{lat}	0.30
5xx Error Ratio	N_{err}	0.20
Queue Length / Backlog	N_{queue}	0.10

$$A_{\text{API}} = 0.4N_{\text{succ}} + 0.3N_{\text{lat}} + 0.2N_{\text{err}} + 0.1N_{\text{queue}}.$$

7.2 Banking APIs

Metric	Symbol	Weight
Success Rate	N_{succ}	0.50
Timeout Ratio	N_{timeout}	0.20
Connection Failures	N_{conn}	0.20
Backlog / Queue Size	N_{queue}	0.10

7.3 Cryptographic Services

Metric	Symbol	Weight
HSM Online State	N_{hsm}	0.40
Success Rate	N_{succ}	0.30
Signing Time	N_{sign}	0.20
Key Slot Availability	N_{slot}	0.10

7.4 Batch Processes

Metric	Symbol	Weight
Completed Within Processing Window	N_{deadline}	0.60
Job Errors	N_{error}	0.20
Backlog / Queue Size	N_{queue}	0.20

7.5 ERP / SAP S/4HANA

Metric	Symbol	Weight
Dialog Response Time	N_{dialog}	0.40
Work Process Utilization	N_{wp}	0.25
Queue Length (SM50/SM66)	N_{queue}	0.20
HANA DB Latency / Locks	N_{hana}	0.15

7.6 ERP / Oracle E-Business Suite

Metric	Symbol	Weight
Active Concurrent Managers	N_{cm}	0.35
Database Wait Events	N_{dbwait}	0.25
Session Pool Usage	N_{sess}	0.20
Workflow Lag	N_{wf}	0.20

7.7 SAP IDoc / EDI Integrations

Metric	Symbol	Weight
IDoc Processing Time (latency)	N_{idoc}	0.45
IDoc Error Ratio (status 51/68)	N_{err}	0.30
IDoc Queue Size	N_{queue}	0.15
Lock Conflicts / DB Blocking	N_{lock}	0.10

7.8 Elasticsearch / OpenSearch (Log-Derived Metrics)

Metric	Symbol	Weight
Log-Derived Error Rate	N_{err}	0.40
Log-Derived Latency (p95/p99)	N_{lat}	0.30
Event Volume Consistency	N_{vol}	0.20
Indexing/Query Lag	N_{lag}	0.10

7.9 Kafka Message Broker

Metric	Symbol	Weight
Consumer Lag	N_{lag}	0.40
Message Throughput (in/out)	N_{rate}	0.30
Produce/Consume Failures	N_{fail}	0.20
Rebalance / Partition Stability	$N_{\text{rebalance}}$	0.10

8 Contour Availability

8.1 Definition of a Contour

In the Unified Availability Model (UAM), a **contour** is defined as a connected set of information systems, services, and processes that together form a *single business flow*, in which a user or an external partner obtains a final business result.

A contour has the following properties:

- **Functional integrity** — all components contribute to one business function;
- **Logical sequence** — systems are called or accessed in a defined order;
- **Technical interdependence** — the failure of one system reduces the availability of the entire contour;
- **Measurability** — each component has a measurable availability value $A_i \in [0, 1]$.

Thus, a contour is not an infrastructure structure by itself, but a **business-oriented chain of dependencies**.

8.2 Contour Composition

Each contour consists of three layers:

1. **Front systems** (front-line layer):
    - API endpoints, web methods, integration gateways;
    - mobile or desktop user interfaces;
    - external REST/gRPC services.
2. **Mid-layer services** (logical layer):
    - CRM/ERP modules;
    - banking APIs, payment processors, billing nodes;
    - cryptographic services (signing, encryption);
    - workflow engines and message queues.
3. **Back-office processes** (background layer):
    - batch jobs;
    - reporting pipelines;
    - scheduled calculations;

- ERP background transactions.

Each component of the contour has its own availability A_i, calculated according to UAM normalization rules.

8.3 Weight Coefficient of a System in a Contour

Every system has a contour weight W_i — the coefficient that reflects its impact on the final operability of the contour.

Weights are determined based on:

- **Component criticality** — can the contour work without this system?;
- **Temporal dependency** — does the system participate in the online path or only in final calculations?;
- **Usage frequency** — how many user or system requests pass through it;
- **Position in the chain** — closer to the user or deeper in the core;
- **Connectivity** — how many other subsystems depend on it.

Formally:

$$\sum_{i=1}^{n} W_i = 1, \qquad W_i \geq 0.$$

This ensures that A_{contour} can be interpreted as the availability of the entire business process.

8.4 Contour Availability Formula

$$A_{\text{contour}} = \sum_{i=1}^{n} W_i A_i,$$

where:

- A_i — availability of a subsystem (normalized in UAM),
- W_i — weight of the subsystem in the contour.

8.5 Example of a Contour Structure

Consider the **payment processing contour**:

- Payment Gateway — 0.25
- Banking API — 0.25

- Document Signing (HSM) — 0.15
- Batch/Reporting (statement generation) — 0.20
- ERP (Oracle EBS / SAP) — 0.15

The contour availability is:

$$A_{\text{pay}} = 0.25A_{\text{gateway}} + 0.25A_{\text{bankAPI}} + 0.15A_{\text{sign}} + 0.20A_{\text{batch}} + 0.15A_{\text{ERP}}.$$

8.6 Why Contour-Level Availability Is More Important

In real business processes:

- a client does not care if ERP is at 99.99% if the banking API works at 80%;
- the failure of even a "non-critical" component (like batch) affects the entire chain;
- weight coefficients allow a fair distribution of influence among components;
- A_{contour} provides management with a business-value indicator rather than a purely technical one.

Therefore, the contour is the **main aggregation level** in the Unified Availability Model.

9 End-to-End Example of Contour Availability Calculation

This section presents a synthetic but realistic example of applying the Unified Availability Model (UAM) to a business process that consists of three heterogeneous systems:

- Web site (frontend + backend),
- Bank–client API (REST banking gateway),
- ERP system SAP S/4HANA (document processing and posting).

Such a contour is typical for scenarios where a user creates a document or payment on a web site, the web service then communicates with the banking API, and the final data is transferred into SAP for further processing.

9.1 Subsystem Metrics

9.1.1 Web Site

Metric	Symbol	Weight
Success Rate	N_{succ}	0.40
Latency $p95$	N_{lat}	0.30
5xx Errors	N_{err}	0.20
DB/Cache Backlog	N_{backlog}	0.10

Table 2: Web site metrics

Observed values:

$$\text{succ} = 98.2\%, \quad p95 = 420\,\text{ms}, \quad 5xx = 1.8\%, \quad \text{backlog} = 250.$$

Baseline / SLA:

$$B_{\text{lat}} = 210\ \text{ms}, \quad SLA = 350\ \text{ms}, \quad L_{\text{max}} = 2000\ \text{ms}, \quad D_{\text{max}} = 1.8\,B_{\text{lat}} = 380.$$

Normalization:

$$N_{\text{succ}} = 0.82, \qquad N_{\text{lat}} = \frac{380}{420} = 0.90,$$
$$N_{\text{err}} \approx 0.60, \qquad N_{\text{backlog}} = 0.75.$$

Final web availability:

$$A_{\text{web}} = 0.40 \cdot 0.82 + 0.30 \cdot 0.90 + 0.20 \cdot 0.60 + 0.10 \cdot 0.75 = 0.79.$$

9.1.2 Bank–Client API

Metric	Symbol	Weight
Success Rate	N_{succ}	0.50
Timeout Ratio	N_{timeout}	0.20
Connection Failures	N_{conn}	0.20
Queue / Backpressure	N_{queue}	0.10

Table 3: Bank–client API metrics

Observed values:

$$\text{succ} = 99.1\%, \quad \text{timeout} = 0.7\%, \quad \text{connfail} = 0.3\%, \quad \text{queue} = 120.$$

Baseline / SLA:

$$B_{\text{succ}} = 99.6\%, \quad SLA_{\text{succ}} = 99\%, \quad SLA_{\text{timeout}} = 1.0\%.$$

Normalization (short form):

$$N_{\text{succ}} = 0.93, \qquad N_{\text{timeout}} = 1, \qquad N_{\text{conn}} = 0.92, \qquad N_{\text{queue}} = 0.85.$$

API availability:

$$A_{\text{api}} = 0.5 \cdot 0.93 + 0.2 \cdot 1 + 0.2 \cdot 0.92 + 0.1 \cdot 0.85 = 0.93.$$

9.1.3 SAP S/4HANA ERP

Metric	Symbol	Weight
Dialog Response Time Issues	N_{dialog}	0.35
HANA DB Wait Events	N_{wait}	0.25
Background Job Lag	N_{lag}	0.20
IDoc/Queue Processing Lag	N_{queue}	0.20

Table 4: SAP S/4HANA metrics

Observed values:

$$\text{dialog} = 1260\,\text{ms}, \quad \text{wait} = 45\,\text{ms}, \quad \text{lag} = 6\,\text{min}, \quad \text{queue} = 240.$$

Baseline / SLA:

$$B_{\text{wait}} = 14\,\text{ms}, \quad SLA_{\text{wait}} = 50\,\text{ms}, \quad T_{\text{max}} = 300\,\text{ms}.$$

Simplified normalization:

$$N_{\text{dialog}} = 0.78, \qquad N_{\text{wait}} = 0.92, \qquad N_{\text{lag}} = 0.88, \qquad N_{\text{queue}} = 0.75.$$

SAP availability:

$$A_{\text{sap}} = 0.35 \cdot 0.78 + 0.25 \cdot 0.92 + 0.20 \cdot 0.88 + 0.20 \cdot 0.75 = 0.83.$$

9.2 Contour Availability Calculation

System weights:

$$W_{\text{web}} = 0.30, \quad W_{\text{api}} = 0.40, \quad W_{\text{sap}} = 0.30.$$

Final contour availability:

$$A_{\text{contour}} = 0.30 \cdot 0.79 + 0.40 \cdot 0.93 + 0.30 \cdot 0.83 = 0.854.$$

9.3 Interpretation

The resulting value

$$A_{\text{contour}} = 0.854$$

indicates a noticeable degradation of the contour, mostly driven by the Web subsystem. The bank–client API is the most stable part of the contour, while SAP S/4HANA shows moderate risk due to elevated queue and job lag.

This example demonstrates the practical applicability of UAM for systems that cannot be combined under a single SLA or metric scale in traditional models.

10 Implementation in Monitoring Systems

The Unified Availability Model (UAM) can be implemented using standard observability platforms without introducing custom software. This section describes practical approaches for Prometheus, Zabbix, and Grafana.

10.1 Prometheus Implementation

Prometheus is the most natural environment for UAM, because normalization, weighting, and aggregation can be implemented directly as PromQL expressions.

Key mechanisms used in UAM

- **Recording rules** for calculating normalized metrics (N_{ij}), system availability (A_i), and contour availability (A_{contour}).
- PromQL functions:
  - `clamp(x, min, max)` — bounding normalized values in [0,1];
  - `rate()` and `irate()` — for latency/error trends;
  - `scalar()` — converting constants into PromQL expressions;
  - `max()`, `min()`, `avg_over_time()` — for baseline calculation.
- **Vector arithmetic** for weighted sums.

Example: Baseline normalization rule

```
record: uam:web_latency_norm
expr: clamp((baseline_lat * 1.8) / web_latency_p95, 0, 1)
```

Example: System availability

```
record: uam:A_web
expr: 0.4 * uam:web_success_norm +
0.3 * uam:web_latency_norm +
0.2 * uam:web_errors_norm +
0.1 * uam:web_backlog_norm
```

Contour availability rule

```
record: uam:A_contour_payment
expr: 0.30 * uam:A_web +
0.40 * uam:A_api +
0.30 * uam:A_sap
```

This produces a real-time contour availability score directly in Prometheus, which can be queried by Grafana or Alertmanager.

Alerting

- Warning: `A_contour < 0.85`
- Major: `A_contour < 0.70`
- Critical: `A_contour < 0.50`

10.2 Zabbix Implementation

Zabbix does not support vector math directly, but UAM can be implemented using:

- **Dependent items** — raw metrics feed "parent items"; normalization formulas are applied in dependent items.
- **User-defined items** — using Zabbix's expression language to compute N_{ij} and A_i.
- **Triggers for availability degradation**
  - Warning: $A_i < 0.8$
  - High: $A_i < 0.6$
  - Disaster: $A_i < 0.5$

Example: Dependent item formula

```
norm_latency =
iif(last("latency_p95") < baseline * 1.8,
1,
(baseline * 1.8) / last("latency_p95")
)
```

Example: System availability item

```
A_web =
0.4 * last("norm_success") +
0.3 * last("norm_latency") +
0.2 * last("norm_errors") +
0.1 * last("norm_backlog")
```

Dashboards

Zabbix dashboards can render:

- Availability trend of systems,
- Weekly availability reports,
- Top-3 degraded systems by UAM score.

10.3 VictoriaMetrics Implementation

VictoriaMetrics is fully compatible with UAM because it supports PromQL and recording rules. It is particularly suitable for large installations where high-cardinality metrics are collected from many heterogeneous systems.

Advantages for UAM

- **High ingestion rate** — suitable for enterprise landscapes with thousands of normalized metrics.
- **Efficient storage** — reduces cost of long-term UAM history (baselines benefit from long retention).
- **Downsampling** — useful for weekly/monthly contour reports.
- **Full PromQL compatibility** — all UAM formulas work without changes.

Baseline computation in VictoriaMetrics

Thanks to low storage cost, VictoriaMetrics allows long retention windows (e.g., 180–365 days), which improves baseline stability.

Example: rolling median baseline:

```
record: uam:web_latency_baseline
expr: quantile_over_time(0.5, web_latency_p95[30d])
```

Seasonal baselines (per hour of day):

```
record: uam:web_latency_hourly_baseline
expr: avg_over_time(web_latency_p95[30d])
unless on(hour()) (vector(1))
```

Contour availability in VictoriaMetrics

Same as in Prometheus:

```
record: uam:A_contour_payment
expr: 0.30 * uam:A_web +
0.40 * uam:A_api +
0.30 * uam:A_sap
```

Alerting

VictoriaMetrics Alert (vmalert) supports the same rules as Alertmanager.

Typical UAM alerts:

- Warning: `A_contour < 0.85`
- Major: `A_contour < 0.70`
- Critical: `A_contour < 0.50`

Dashboards

VictoriaMetrics integrates seamlessly with Grafana, so the visualization layer described in previous sections applies without modifications.

10.4 Loki and Log-based Normalization

Loki provides log aggregation and indexing, and although Loki does not support numeric vector math like PromQL, it can supply raw event counts that feed into UAM normalization.

Loki is especially useful for systems where:

- latency is not instrumented as a metric,
- errors exist only in logs,
- business events (e.g., “payment posted”, “HSM slot error”) appear only in log streams.

Using Loki for UAM Metrics

Loki queries (LogQL) can be transformed into numeric metrics using:

- `count_over_time()` — error/event rates;
- `rate()` — sliding-window frequency;
- `sum by(...)` — aggregation across pods/components;
- `label_replace()` — mapping log fields to metric labels.

Example: extract 5xx-equivalent failures from logs:

```
sum(rate(({app="web"} |= "HTTP 5") [5m]))
```

Transform to normalized error ratio:

```
record: uam:web_errors_norm
expr: clamp(1 - error_rate / 0.02, 0, 1)
```

Extracting Business Events

Example: normalize “payment posted” success ratio from logs:

```
payment_success =
sum(rate(({job="sap"} |= "PAYMENT_POSTED") [5m]))

payment_failure =
sum(rate(({job="sap"} |= "PAYMENT_FAILED") [5m]))

record: uam:sap_payment_norm
expr: clamp(payment_success /
(payment_success + payment_failure), 0, 1)
```

Integrating Loki with Prometheus or VM

Loki metrics are usually exported via:

- **Promtail → Prometheus**, or
- **Loki ruler → VictoriaMetrics**.

These derived metrics then participate in the standard UAM formulas for:

$$A_i = \sum_j w_{ij} N_{ij}$$

and

$$A_{\text{contour}} = \sum_i W_i A_i.$$

Loki Dashboards in Grafana

Recommended UAM-oriented panels:

- Error-rate log timeline (for N_{err}),
- Business-event throughput,
- Derived metric gauges,
- Correlation panels ("error spikes → SAP delay → decreased A_{sap}").

Loki adds value where metric-based observability is incomplete — it closes the gap between logs and normalized metric availability.

10.5 Grafana Visualization

Grafana provides the most expressive layer for UAM. It displays normalized metrics, system-level availability, and contour values.

Recommended Panels

- **Gauge panels** for A_i and A_{contour}.
- **Composite SLA panels** — combining latency, errors, and success rate in one view.
- **Time-series panels** for each normalized metric N_{ij}.
- **Status blocks** for top priority systems.

Four-level color logic

- **Green** — $A \geq 0.90$ (healthy)
- **Yellow** — $0.80 \leq A < 0.90$ (minor degradation)
- **Orange** — $0.60 \leq A < 0.80$ (major impact)
- **Red** — $A < 0.60$ (critical)

Contour dashboard layout

- Row 1: **Contour availability** (large gauge)
- Row 2: Three system gauges (Web, API, SAP)
- Row 3: Drill-down to N_{ij} metrics for each subsystem
- Row 4: Raw metrics (latency p95, timeouts, wait events)

This structure gives both technical engineers and managers a clear picture of where the degradation originates.

11 Conclusion

The Unified Availability Model (UAM) provides a formal and extensible approach for calculating the availability of heterogeneous information systems. The methodology makes it possible to:

- compare different classes of applications in a fair and consistent way;
- build a unified availability indicator for an entire business-process contour;
- integrate the model with Prometheus, Zabbix, Grafana, and other observability tools;
- adapt the framework to any new system type without changing its core principles.

UAM is intended for practical use in high-load and business-critical IT landscapes, where systems differ significantly in architecture, performance characteristics, and operational behaviour.

Abstract to Appendix A

Appendix A reviews alternative approaches to aggregating metrics in heterogeneous IT systems. It explains why fuzzy logic is applicable **only within** individual levels (Infra, App, Biz), where boundaries of “normal” behaviour are inherently vague, and why its use **between levels** is

unacceptable due to strict hierarchical dependency: an upper level cannot be more available than the level it depends on.

The appendix also justifies the need for dynamic reweighting, introduces soft and hard hierarchical coherence models, and briefly analyses why neural-network-based approaches are unsuitable due to retraining requirements and low interpretability.

This appendix provides the methodological foundation for UAM v2 and formalises the engineering logic behind a multi-layer availability model.

Appendix A. Fuzzy Logic and Neural Networks in Availability Assessment: Scope of Application and Limitations

Note. Appendix A is introduced in Version 2.0 and formalises the methodological justification for hierarchical coherence and fuzzy-logic constraints.

A.1. Why fuzzy logic is acceptable *within* levels (Infra / App / Biz)

A.1.1. Motivation

Low-level metrics often have vague or ambiguous definitions of "normal". Typical examples include:

- daytime and nighttime latency may differ by a factor of 2–3 while still being considered normal;
- acceptable ping values may vary across operational teams;
- backlog thresholds cannot be unified across workload types.

General problem: **the boundaries between "good", "normal", and "bad" are interval-based rather than strict.**

Fuzzy logic applies naturally when there are:

- soft or shifting thresholds,
- expert-driven assessments,
- ambiguous operating regimes.

Thus, fuzzy logic resolves primarily an **epistemic (human-interpretive) problem**, not a mathematical one.

A.1.2. Benefits of fuzzy logic inside a level

- **Reduction of subjective debates.** Ambiguous thresholds become explicit formal ranges.
- **Smooth response.** Values change continuously rather than jumping.
- **Noise resistance.** Small fluctuations do not trigger false alarms.
- **Better preprocessing.** Fuzzy logic is applied before computing N_{ij}.
- **Flexibility without loss of control.** UAM remains strict; fuzzy logic is only a normalisation tool.

Thus, fuzzy logic is applicable strictly as an **intra-level smoothing mechanism** for Infra, App, and Biz metrics.

A.2. Why fuzzy logic must *not* be used *between* levels (Infra $\rightarrow$ App $\rightarrow$ Biz)

A.2.1. Motivation for rejection

Level hierarchy is characterised by strict dependency:

- infrastructure degradation inevitably degrades applications;
- application unavailability blocks business operations;
- an upper level cannot exceed the availability of the lower one.

Fundamental principle:

the coherence of an upper level is bounded by the coherence of the lower level.

Applying fuzzy logic between levels leads to:

- blurred dependencies,
- contradictory evaluations,
- a risk of "healthy-looking Biz" while Infra is degraded,
- loss of interpretability,
- recursive instability in recalculations.

Fuzzy logic effectively makes the levels "too equal", which is **structurally incorrect** for a dependency hierarchy.

A.2.2. Benefits of rejecting fuzzy logic between levels

- **Strict hierarchical constraints.** A level cannot override the limitations of the level below.
- **Business-transparent semantics.** Managers do not need to understand fuzzy-membership values.
- **Correct propagation of degradations.** If Infra is red, App and Biz cannot be green.
- **Honest availability model.** $\min(\cdot)$ and multiplicative operators maintain realism.
- **Compatibility with UAM and COE principles.**
- **Elimination of recursive instability.** Inter-level fuzzy logic causes uncontrolled renormalisation.

A.3. Why neural networks are not used

Neural-network-based prediction and aggregation models were evaluated, but intentionally rejected.

Reasons:

- changes in workload profiles require **full retraining**, which is unacceptable for a governance-level availability metric;
- neural networks have *low interpretability*;
- strict repeatability and auditability are required, which is impossible with stochastic weights;
- neural networks optimise prediction, not SLA responsibility.

Thus, neural networks are unsuitable for **executive-level availability metrics**.

A.4. Conclusion

Fuzzy logic may be applied only for normalising individual metrics within levels (Infra, App, Biz) — and **must not be used** for aggregating levels.

This is an intentional engineering decision:

- metrics have fuzzy boundaries → fuzzy is appropriate;
- levels form a strict hierarchy → fuzzy is inappropriate.

References

[1] A.A. Nekludoff, *Coherent Observational Epistemology: Foundational Principles, Secondary Principles, and Axiomatic System*, Zenodo (2025). doi:10.5281/zenodo.17646288.

[2] A.A. Nekludoff, *Philosophy of Discrete Being: The Foundational Manifesto*, Zenodo (2025). doi:10.5281/zenodo.17646888.

[3] A.A. Nekludoff, *Unified Availability Model (UAM): Методика расчёта доступности разнородных ИТ-систем*, Zenodo (2025). doi:10.5281/zenodo.17721164.

Version History

Version	Description
v1.0 (2025-11-25)	Initial release of the Unified Availability Model (UAM). Introduced the core framework: metric normalization, system-level availability, and contour-level aggregation. Included baseline/SLA normalization rules and examples for API, batch, ERP, cryptographic services, and SAP/IDoc integrations.
v2.0 (2025-11-26)	Revised and extended edition. Added the Hierarchical Coherence Model (HCM) for multi-level availability, including: • soft multiplicative coherence model, • strict foundational limit model ($H_{\text{total}} = \min(\cdot)$), • dynamic re-weighting of metrics within levels. Added Appendix A on fuzzy-logic applicability and neural-network limitations. Refined definitions, improved terminology, expanded examples, and added cross-system applicability notes.