Information, Computing and Intelligent Systems, 2026, No. 8, 113 – 133
UDC 004.41:004.272.43 https://doi.org/10.20535/2786-8729.8.2026.361321

FORMAL MODEL OF ERROR ESCALATION AND RECOVERY IN MULTILAYER ASYNCHRONOUS ARCHITECTURES WITH AN EVENT LOOP

Dmytro Nechai*
https://orcid.org/0000-0002-0696-9985

Timur Shemsedinov
https://orcid.org/0000-0001-5958-4731

National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine

*Corresponding author: nechaido@gmail.com

Received: 27 April 2026 / Accepted: 11 May 2026 / Published: 27 May 2026

Layered asynchronous architectures with an event loop combine intra-process scheduling, worker-thread execution, child-process isolation, and network interaction. Their reliability depends on how failures are represented, escalated, recovered, and converted at architectural boundaries. The object of this study is error escalation and recovery in such architectures. The purpose is to increase the reliability of layered asynchronous systems by developing and empirically evaluating a formal model that links error representations, escalation boundaries, recovery strategies, and overhead within the Imperative Shell/Multi-Paradigm Core pattern. The study uses formal comparison of error-representation contracts, boundary analysis of event-loop, worker-thread, child-process, and network communication, architectural modelling, and reproducible microbenchmarking on the Node.js platform. The proposed model includes a taxonomy of four error representations: thrown exceptions, error-first callbacks, the *Result monad*, and typed error codes. It also defines escalation semantics for event-loop phases and process, worker, and network boundaries. The model assigns total value-based computation to the multi-paradigm core and side-effectful recovery, logging, retry, circuit breaking, compensation, supervision, and protocol conversion to the imperative shell. Empirical evaluation shows that thrown errors are hundreds of times more expensive than value-typed alternatives, that retry and circuit-breaker wrappers add only tens of nanoseconds on the successful path, and that worker restart latency is about 20 ms. The scientific novelty lies in combining representation taxonomy, boundary-sensitive escalation semantics, recovery-strategy placement, and measurable overhead into one formal model. The obtained recommendations support the design of fault-tolerant Node.js systems.

Keywords: error escalation, asynchronous architecture, event loop, error representation, recovery strategy, Result monad, supervision tree, fault tolerance, JavaScript, Node.js.

1. Introduction

Modern industrial systems are built as layered asynchronous architectures. Intra-process event loops serve thousands of concurrent requests, worker-thread pools execute compute-heavy tasks, and network layers compose dozens of independent services. The topic of error handling in such systems is highly relevant, since errors do not stay local. Errors escalate across several execution layers, cross serialization boundaries, and accumulate into failure cascades. An analysis of 198 production failures across distributed data-intensive systems found that 92% of catastrophic failures arise from incorrect handling of non-fatal errors that were explicitly signalled in software [1]. Industry surveys further report that one hour of unplanned downtime costs over USD 300 000 in the majority of mid-to-large enterprises [2], which makes principled error escalation an economically and operationally load-bearing concern. A formal understanding of error escalation is therefore a necessary prerequisite for building reliable systems.

Coordination primitives ensure correct operation when things go right: mutexes serialize access, Conflict-free Replicated Data Types (CRDTs) converge replicated state, and actors process messages

sequentially. However, production systems spend significant complexity budgets on what happens when things go wrong. Such scenarios include an unhandled rejection in a `Promise` chain, a worker thread that crashes mid-operation, and a network request that completes by timeout after a partial side effect.

Error handling in asynchronous architectures is fundamentally different from synchronous exception propagation. Stack unwinding in a synchronous call stack yields predictable delivery of an exception to the nearest `catch`. In an asynchronous system, errors cross three boundaries with distinct escalation semantics. First, `await` points within a single event loop, where `Promise` rejection chains replace stack unwinding. Second, `MessagePort` boundaries between worker threads, where structured cloning cannot transfer `Error` objects faithfully. Third, network boundaries, where errors manifest as timeouts, partial responses, or connection resets, with no guaranteed notification.

The above examination of the topic supports the conclusion that, without a formal account of error escalation, it is not possible to design layered asynchronous systems with predictable fault tolerance in a systematic manner. Therefore, developing and empirically evaluating a formal model of error escalation and recovery is both appropriate and timely. Such a model must explain how errors change their representation and recovery mechanism when execution crosses event-loop, worker-thread, child-process, and network boundaries.

2. Literature review and problem statement

A survey of recent literature reveals several research strands that are pertinent to error handling in asynchronous systems. The first strand covers the formal semantics of exceptions and of computations that may fail. The work [3] shows that adding exceptions to a programming language requires reconciliation of invariants across every abstraction through which an exception may propagate. The study [4] argues that monadic composition is a natural way to represent failing computations as values, which allows static tracking of error escalation across function signatures. These works, however, consider synchronous language models and do not address escalation across event loops or message channels.

The second strand covers models of asynchronous execution. The paper [5] argues for the advantages of the actor model for fault-tolerant systems, and the study [6] describes how a contract-bound `Promise` model enables asynchronous composition. The study [7] gives a formal denotational semantics of asynchronous JavaScript, including microtask and macrotask scheduling rules that are also fixed by the HTML Living Standard [8]. The work [9] develops a small-step operational model of JavaScript `Promise` objects together with a typing discipline that captures common misuse patterns. In all of these sources, however, the rejection-escalation mechanism is treated only at the level of base semantics and is not linked to architectural patterns for post-failure recovery.

The third strand covers recovery strategies and fault tolerance. The work [10] introduces the *CircuitBreaker* pattern as an architectural mechanism for preventing cascading failures. The paper [11] formulates the model of long-running transactions known as *Saga*, with compensating actions. The doctoral dissertation [12] develops the Erlang/OTP supervision model in which failure is localized by restart rather than by handling at the site of occurrence. Industrial sources describe associated patterns such as request retry, timeouts, and bulkheads, but a systematic comparison of cost, prerequisites, and applicability for event-loop runtimes is absent from the literature.

The fourth strand covers architectural models for distributed applications. The paper [13]] formalizes a CRDT for replicated JSON documents and quantifies the cost of replication semantics, and the studies [14, 15] analyze CRDTs more broadly as a mechanism for automatic reconciliation of conflicting updates. Consistency and error escalation are nevertheless treated separately: CRDT research concentrates on state convergence, not on how message-delivery failures surface at the application layer.

The fifth strand covers the architectural pattern that separates a pure core from an imperative shell. The work [16] argues that confining side effects to the application boundary simplifies testing

and formal reasoning about correctness. The study [17] analyses the semantics of asynchronous events in JavaScript and proposes a formal event-loop model that is suitable for static analysis. Even so, the question of where error boundaries lie within such an architecture, and which error representation each layer should adopt, has not received systematic treatment.

The above analysis supports the following generalization. Formal models of exceptions cover synchronous semantics; models of asynchronous execution cover nominal behaviour; individual recovery strategies are described outside a unified taxonomy; and architectural patterns are not connected to a quantitative assessment of strategy cost. The unsolved scientific problem is the absence of a unified formal model of error escalation and recovery in layered asynchronous architectures with an event loop. Existing studies separately describe synchronous exceptions, asynchronous scheduling, recovery patterns, and architectural separation of side effects. However, they do not explain these elements as one model that accounts for escalation boundaries, representation fidelity, recovery placement, and measurable overhead. This problem justifies a study focused on the development and empirical evaluation of such a model.

3. The aim and objectives of the study

The aim of the study is to increase the reliability of layered asynchronous architectures with an event loop by constructing a formal model that unifies error representations, boundary-specific escalation semantics, recovery strategies, and placement under the Imperative Shell/Multi-Paradigm Core architectural pattern, and by quantifying the associated overhead through reproducible measurement on the Node.js platform.

To achieve this aim, the following objectives are set.

1. To develop a taxonomy of error representations used in event-loop runtimes, characterizing each representation by its value domain, signalling mechanism, and escalation rule, and by the trade-offs in composability, signature visibility, and cross-boundary fidelity.

2. On the basis of the developed taxonomy, to construct a formal model of error escalation and recovery, comprising: escalation semantics across event-loop phases and across worker-thread, child-process, and network boundaries; contracts for the principal recovery strategies – *retry with exponential back-off*, *Circuit Breaker*, *Saga* compensation, and supervision trees; and placement rules for representations and strategies under the Imperative Shell/Multi-Paradigm Core pattern.

3. To empirically evaluate the proposed model on the Node.js platform by measuring the cost of error representations, cross-boundary serialization, and recovery-strategy overhead under a reproducible benchmark harness, and on this basis to formulate quantitative recommendations for the choice of error representation and recovery strategy at each architectural layer.

4. Materials and methods for developing and evaluating the formal model of error escalation and recovery

4.1. Object, subject, and hypotheses of the study

The object of the study is error escalation and recovery in layered asynchronous architectures with an event loop. The subject of the study is the formal model that describes error representations, escalation boundaries, recovery strategies, and their quantitative characteristics.

The main hypothesis of the study is that a boundary-sensitive formal model can increase the reliability of layered asynchronous architectures by assigning error representations and recovery strategies according to escalation boundary, data-fidelity requirements, and measured overhead. The following working hypotheses are advanced.

(H1) Value-typed error representations, including `Result.error` and typed code objects, are at least an order of magnitude cheaper than instances of the `Error` class created by `throw` because they do not require stack-trace capture.

(H2) The main recovery strategies for layered Node.js architectures are *retry with exponential back-off*, *Circuit Breaker*, *Saga* compensation, and *supervision trees*, because they cover transient dependency failures, dependency degradation, distributed multi-step failures, and worker or process crashes.

(H3) The *Imperative Shell/Multi-Paradigm Core* pattern can serve as an architectural placement rule for the model: the imperative shell becomes the recovery and escalation boundary, while the multi-paradigm core remains a zone of total or value-returning computations.

(H4) Worker-thread restart latency on contemporary hardware is on the order of tens of milliseconds, which makes structured supervision practical for workers whose useful work is longer than the restart interval.

4.2. Taxonomy of error representations

Definition 1 (*Error representation*). An error representation is defined as the triple

$$E = (D, S, P), \tag{1}$$

where D is a domain of error values, S is a signalling mechanism that describes how an error is produced, and P is an escalation rule that specifies how an unhandled error moves through the system.

Four representations dominate the JavaScript/Node.js ecosystem, and are described below.

Exceptions (`throw/try-catch`). The domain contains any value, typically a subclass of `Error`. Signalling proceeds through the `throw` statement, which interrupts control flow. The escalation rule combines synchronous exception propagation (stack unwinding) with asynchronous escalation across `Promise.reject`. The caller is required to wrap the call site in `try-catch` for the synchronous portion or in `try-catch` around an await expression. An `unhandledrejection` raises the process-level `unhandledRejection` event. The following code fragment illustrates a typical use of exceptions:

```
const divide = (a, b) => {
    if (b === 0) throw new RangeError('Division by zero');
    return a / b;
};

const safeDivide = async (a, b) => {
    if (b === 0) throw new RangeError('Division by zero');
    return a / b;
};
```

Trade-offs: exceptions provide natural control-flow interruption and high-quality stack traces for debugging. However, they remain invisible in function signatures, couple error detection with error handling inside a single catch block, and compose poorly in a functional style.

Error-first callbacks (the Node.js convention). The domain is the union `Error | null` used as the first argument. Signalling occurs through invocation of the callback with the pair (`error`, `result`). Escalation is manual: the caller inspects the error before reading the result. This pattern historically dominated the core Node.js API but has been largely superseded by `Promise` and `async-await`.

The *Result/Either monad*. The domain is the tagged union `Ok(value) | Err(error)`. Signalling occurs through the return value, without interrupting control flow. Escalation is monadic: errors escalate across the map and `chain` operators without explicit checks. The following code fragment illustrates a typical implementation:

```
class Result {
    #value = null;
    #error = null;
```

```
        constructor(value, error) {
            this.#value = value;
            this.#error = error;
        }

        static ok(value) {
            return new Result(value, null);
        }

        static error(error) {
            return new Result(null, error);
        }

        map(fn) {
            return this.#error ? this : Result.ok(fn(this.#value));
        }

        chain(fn) {
            return this.#error ? this : fn(this.#value);
        }

        match({ ok, error }) {
            return this.#error ? error(this.#error) : ok(this.#value);
        }
    }
```

The contract states that the return type is always an instance of the `Result` class. The caller chooses the handling path through the `map/chain` operators or through the `match` combinator. Trade-offs: the monad composes; errors are visible in types; there is no hidden control flow. The cost is caller discipline and an extra object allocation.

Typed error codes. The domain comprises enumerated or string constants with associated data. Signalling occurs through the return of an object of the form `{ code, data }` or `{ code, error }`. Escalation reduces to explicit pattern matching on the `code` field. This approach survives serialization naturally and is therefore the preferred option for cross-boundary communication. The following code fragment illustrates a typical implementation:

```
    const parsePort = (input) => {
        const port = parseInt(input, 10);
        if (Number.isNaN(port)) {
            return { code: 'INVALID_FORMAT', input };
        }
        if (port < 1 || port > 65535) {
            return { code: 'OUT_OF_RANGE', port };
        }
        return { code: 'OK', port };
    };
```

A summary comparison of the four representations across the principal properties is given in Table 1.

The data in Table 1 show that no single representation is optimal across every criterion. Visibility in the signature and composability are obtained at the cost of natural control-flow interruption and

Table 1. Comparison of error representations by principal properties

Property	Exceptions	Error-first callback	Result monad	Error codes
Composability	Low	Low	High	Medium
Visibility in signature	Hidden	Visible (convention)	Visible (type)	Visible
Cross-boundary fidelity	Poor	Medium	Medium	Excellent
Control-flow interruption	Yes	No	No	No
Debugging via stack	Excellent	Poor	Medium	Poor
Paradigm affinity	Imperative	Procedural	Functional	Any

stack-trace quality, which motivates the layered hybrid use of representations described in Section 4.6.

4.3. Error escalation across event-loop phases

Synchronous exception propagation (stack unwinding) applies inside synchronous regions of JavaScript execution. The single-threadedness invariant of the `event loop` guarantees that no interleaving occurs during unwinding, so the error reaches the nearest `catch` block or becomes an unhandled error at the process level. `Promise` rejections escalate across the microtask queue. An unhandled rejection raises the `unhandledRejection` event at the end of the current microtask-drain cycle. The following code fragment illustrates a typical handler:

```
process.on('unhandledRejection', (reason, promise) => {
    const event = { reason, promise };
    logger.error('Unhandled rejection', event);
});
```

Proposition 1 (*Rejection visibility*). A `Promise` rejection is observable as an `unhandledRejection` event if and only if no `.catch()` handler or `try-catch` block around an `await` is attached by the time the microtask queue is drained. This dependency on timing implies that late-attached handlers may miss the event.

Errors raised inside timer callbacks (`setTimeout`, `setInterval`) and input-output callbacks escalate independently of the call site. Each macrotask executes in its own protected frame, so an unhandled error in one callback does not affect another.

Node.js streams surface errors through the `'error'` event. An instance of the `EventEmitter` class with no listener on `'error'` throws synchronously inside `emit('error')`, which terminates the process by default.

Proposition 2 (*EventEmitter error invariant*). An instance of the `EventEmitter` class that emits the `'error'` event with zero listeners throws synchronously in the current execution context, which is equivalent to the `throw` statement. This is the only `EventEmitter` event with special semantics.

4.4. Cross-boundary error escalation

Node.js worker threads communicate through `MessagePort`. Error escalation has three channels. The first channel is the `'error'` event on the `Worker` object, which is raised when the main module of the worker throws an exception. The `Error` instance is serialised with the stack trace preserved, while own properties and the prototype chain are lost. The second channel is the `'messageerror'` event, which is raised when a received message cannot be deserialised. The third channel is the `'exit'` event with a non-zero code, which indicates a worker crash or a call to `process.exit()`. The following code fragment illustrates a typical registration of listeners:

```
const worker = new Worker('./task.js');
worker.on('error', (error) => {
    logger.error('Worker error', { error });
});
worker.on('exit', (code) => {
    if (code !== 0) handleCrash(code);
});
```

A separate problem is the absence of an error channel for the shared-memory buffer `SharedArrayBuffer`. If a worker crashes while holding an atomic lock, that lock is never released, which leads to a deadlock. Implementations of the Web Locks API resolve this issue through timeout-based recovery and an option to steal an orphaned lock.

Child-process errors (`child_process`) escalate across the standard error stream stderr as text, across process exit codes as integers, and across the `'error'` event when spawn failures occur. Error fidelity is lower than for worker threads, since structured error objects are unavailable and only text streams and numeric codes remain.

Network errors manifest as connection refusal, timeout, partial response, HTTP status codes, and protocol errors. Unlike process boundaries, network errors do not guarantee notification: a remote process may fail silently, and the caller learns of the failure only through timeout expiry. A summary of error fidelity by boundary is given in Table 2.

Table 2. Characteristics of error-escalation boundaries

Boundary	Error fidelity	Notification guarantee	Recovery signal
Intra-loop (`await`)	Full (`Error` object)	Immediate	Rejection / `throw`
Worker thread	Partial (serialised)	On crash	`'error'` / `'exit'` events
Child process	Low (text / code)	On exit	`stderr` / exit code
Network	Minimal (timeout / status)	Not guaranteed	Timeout / HTTP status

The data of Table 2 indicate that, as the boundary becomes more remote, the fidelity of the error decreases monotonically and the notification guarantee weakens. This observation determines the choice of error representation. The further from the process core, the less practical it is to rely on a full-fidelity `Error` instance, and the more appropriate it becomes to use a serialization-safe error code.

4.5. Recovery strategies

Retry with exponential back-off. The simplest recovery strategy consists of re-executing the operation with growing delays between attempts. This approach is appropriate for transient failures, for example short-lived network disturbances or temporary unavailability of a dependency. The following code fragment illustrates a typical implementation:

```
const retry = async (fn, options = {}) => {
    const maxAttempts = options.maxAttempts ?? 3;
    const baseDelay = options.baseDelay ?? 100;
    const factor = options.factor ?? 2;
    for (let attempt = 1; attempt <= maxAttempts; attempt++) {
        try {
            return await fn();
        }   catch (error) {
```

```
            if (attempt === maxAttempts) throw error;
            const delay = baseDelay * factor ** (attempt - 1);
            const jitter = delay * (0.5 + Math.random() * 0.5);
            await new Promise((r) => setTimeout(r, jitter));
        }
    }
};
```

The contract requires the function `fn` to be idempotent, that is, repeated execution must produce the same observable effect. Non-idempotent operations require deduplication by an idempotency key or protection on the receiver side.

Circuit breaker (`CircuitBreaker`). This pattern prevents cascading failures by short-circuiting calls to a failing dependency. Three states are defined: `closed` – nominal operation with failure counting; `open` – every call is rejected immediately during a cool-down interval; `half-open` – a single trial call is permitted in order to test recovery. The following code fragment illustrates a typical implementation:

```
class CircuitBreaker {
    #state = 'closed';
    #failures = 0;
    #nextRetry = 0;
    #threshold = 5;
    #cooldown = 30000;

    constructor({ threshold = 5, cooldown = 30000 } = {}) {
        this.#threshold = threshold;
        this.#cooldown = cooldown;
    }

    async call(fn) {
        if (this.#state === 'open') {
            if (Date.now() < this.#nextRetry) {
                throw new Error('Circuit open');
            }
            this.#state = 'half-open';
        }
        try {
            const result = await fn();
            this.#onSuccess();
            return result;
        } catch (error) {
            this.#onFailure();
            throw error;
        }
    }

    #onSuccess() {
        this.#state = 'closed';
        this.#failures = 0;
    }

    #onFailure() {
```

```
            this.#failures++;
            const trip = this.#failures >= this.#threshold || this.#state === 'half-open';
            if (trip) {
                this.#state = 'open';
                this.#nextRetry = Date.now() + this.#cooldown;
            }
        }
    }
```

The contract requires the user to configure two parameters: the failure threshold for tripping and the cool-down duration. The state of the breaker is process-local, so multi-instance deployments require coordination of state through shared storage.

Saga compensation. For distributed operations across multiple services, the *Saga* model defines a sequence of transactions with compensating actions

$$\big[(T_1, C_1), (T_2, C_2), \ldots, (T_n, C_n)\big]. \tag{2}$$

If transaction

$$T_k, \tag{3}$$

fails, the compensations

$$C_{k-1}, \ldots, C_1 \tag{4}$$

execute in reverse order. The following code fragment illustrates a typical implementation:

```
const executeSaga = async (steps) => {
    const completed = [];
    for (const { execute, compensate } of steps) {
        try {
            const result = await execute();
            completed.push({ compensate, result });
        } catch (error) {
          for (const step of completed.reverse()) {
              try {
                  await step.compensate(step.result);
              }   catch (exception) {
                  logger.error('Compensation failed', { exception });
              }
          }
          throw error;
        }
    }
};
```

The contract requires every step to provide two functions: a forward action and a compensating action. Compensations must be idempotent, since they may be retried. Compensations themselves may fail, so a production implementation requires dead-letter queues or manual-intervention hooks.

Supervision tree. The Erlang/OTP (Open Telecom Platform) philosophy organizes processes into trees in which a supervisor monitors children and applies restart strategies. This pattern is adapted to Node.js worker threads. The following code fragment illustrates a typical implementation:

```
class Supervisor {
    #children = new Map();
```

```
    #strategy = 'one-for-one';
    #maxRestarts = 3;
    #window = 60000;
    #restartLog = [];

    constructor(options = {}) {
        this.#strategy = options.strategy ?? 'one-for-one';
        this.#maxRestarts = options.maxRestarts ?? 3;
        this.#window = options.window ?? 60000;
    }

    register(name, workerPath, options = {}) {
        const start = () => {
            const worker = new Worker(workerPath, options);
            worker.on('exit', (code) => {
                if (code !== 0) this.#handleCrash(name);
            });
            worker.on('error', (error) => {
                logger.error(`Worker ${name}`, { error });
            });
            this.#children.set(name, {
                worker,
                workerPath,
                options,
                start,
            });
        };
        start();
    }
    #handleCrash(name) {
        const now = Date.now();
        this.#restartLog = this.#restartLog.filter(
            (t) => now - t < this.#window,
        );
        this.#restartLog.push(now);
        if (this.#restartLog.length > this.#maxRestarts) {
            logger.error('Max restarts exceeded, escalating');
            return;
        }
        if (this.#strategy === 'one-for-one') {
            const child = this.#children.get(name);
            if (child) child.start();
        } else if (this.#strategy === 'one-for-all') {
          for (const [, child] of this.#children) {
              child.worker.terminate();
              child.start();
          }
        }
    }
}
```

Three restart strategies are supported. The `one-for-one` strategy restarts only the failed worker, which is the natural choice for independent workers. The `one-for-all` strategy restarts every child, which is appropriate for interdependent workers with shared state. The `rest-for-one` strategy restarts the failed worker together with all workers registered after it, which matches ordered dependencies.

Proposition 3 (*Bounded restart*). A supervisor configured with `maxRestarts` = m and `window` = w guarantees that at most m restarts occur in any interval of length w. If the limit is exceeded, the supervisor escalates the failure up the hierarchy or stops the subsystem.

A summary of recovery-strategy trade-offs is given in Table 3.

Table 3. Recovery strategies and their applicability

Strategy	Failure type	Prerequisite	Overhead
Retry + back-off	Transient	Idempotent operation	Low
Circuit Breaker	Dependency degradation	Fast failure preferred	Low
Saga compensation	Multi-step distributed	Compensations defined	Medium
Supervision tree	Worker / process crash	Stateless or recoverable workers	Medium

The data of Table 3 show that no recovery strategy is universal. Selection is driven by the type of failure and by the constraints of the operation, which motivates the layered combination described in Section 4.6.

4.6. Integration with the Imperative Shell / Multi-Paradigm Core architecture

The Imperative Shell/Multi-Paradigm Core pattern confines side effects to the shell. Error handling follows the same structure. The pure core contains total functions that return values of the `Result` class rather than throwing exceptions. The core therefore does not require an error-escalation mechanism; errors themselves are values inside functional composition. The imperative shell is the boundary layer that catches exceptions raised by input-output operations and converts them into values of the `Result` class or into typed error codes. The shell also applies recovery strategies and converts errors into protocol responses for external consumers.

In this study, the pattern is interpreted not only as a separation of side effects from pure computation but also as a separation of error responsibilities. The imperative shell owns interaction with input-output, timers, workers, child processes, network protocols, logging, retries, *Circuit Breakers*, compensation, and supervision. These operations are allowed to fail because they depend on external state. Therefore, the shell is responsible for catching, normalizing, escalating, recovering, and converting errors into protocol-level responses.

The multi-paradigm core owns deterministic domain computation. It may combine functional, object-oriented, procedural, and data-oriented techniques, but its error contract remains value-based. Expected domain failures are returned as `Result` values or typed codes. They are not thrown as exceptions. This makes the core easier to test, compose, and analyze because error cases are represented in function results rather than hidden in control flow.

The boundary between the shell and the core is therefore also the main semantic boundary of the proposed model. Exceptions are acceptable inside the shell when stack traces are consumed for diagnostics. `Result` values and typed codes are preferred inside the core and at serialization boundaries. Recovery strategies are not placed inside domain computations. They are applied by the shell around operations that communicate with unstable external resources.

The following code fragment illustrates a validator in the pure core together with a boundary request handler in the shell:

```
const validatePort = (input) => {
    const port = parseInt(input, 10);
    if (Number.isNaN(port)) {
        return Result.error({ code: 'INVALID_FORMAT', input });
    }
    if (port < 1 || port > 65535) {
        return Result.error({ code: 'OUT_OF_RANGE', port });
    }
    return Result.ok(port);
};

const handleRequest = async (req) => {
    const portResult = validatePort(req.query.port);
    return portResult.match({
        ok: (port) => startService(port),
        error: (e) => ({ status: 400, body: e }),
    });
};
```

A summary recommendation for the choice of error representation by architectural layer is given in Table 4.

Table 4. Error representation by architectural layer

Layer	Representation	Rationale
Pure core	*Result monad*	Composable; total functions; no hidden control flow
Intra-process	Exceptions + `try-catch`	Natural for input-output; stack trace; familiar
Inter-process	Typed codes	Safe for serialization; survive structured cloning
Distributed	Error codes + status	Minimal fidelity; timeout as primary signal

The data of Table 4 imply a four-tier hierarchy for error handling. The first tier – the core – eliminates error escalation through totality of functions. The second tier – intra-process – relies on language-level exception mechanisms. The third tier – inter-process – bridges process boundaries through supervision and serialization-safe codes. The fourth tier – distributed – accepts unreliable delivery and relies on redundancy and timeouts. This distribution of representations across tiers reconciles the cost of error handling with the strength of the guarantees that each boundary provides.

4.7. Experimental procedure

For the empirical verification of hypotheses H1, H2, H3, and H4, an open-source benchmark harness was developed and is included with the article as supplementary material in the directory `benchmarks/`. The measurements reported in Tables 5–6 were obtained on the following environment, fixed in the file `benchmarks/results/latest.json`:

- runtime: Node.js 22.22.2, V8 engine 12.4.254.21-node.39, libuv library 1.51.0;
- operating system: macOS 26.3 (build 25D125), kernel Darwin 25.3.0;
- processor: Apple M1 Pro, 8 cores, base frequency 2400 MHz, arm64 architecture (little-endian);
- main memory: 16 GB.

The harness is written in plain JavaScript without external dependencies. Each micro-benchmark is warmed up for one tenth of the measured iteration count to let V8 enter the TurboFan optimization

tier. After warm-up, the measured loop runs with a forced garbage-collection pass enabled by the `--expose-gc` flag.

For hypothesis H1, the iteration count is 5 million for synchronous exception operations, 2 million for `Promise` operations, and 20 million for value-typed operations. For the decomposition of exception cost in Section 5.3, the iteration count is 1 million for the call-stack-depth sweep and 5 million for each subclass and for the stack-trace-limit variant. For the cross-boundary serialization measurements in Section 5.4, the iteration count ranges from 500 thousand for `structuredClone(Error)` to 10 million for `structuredClone(string)`.

For the strategies of hypothesis H2, the iteration count is 2 million for paths without delay and 200 for the retry path with three attempts; the jitter parameter is disabled, which makes the total delay deterministic. For hypothesis H4, a total of 300 complete worker-spawn cycles are performed; the worker restart is measured as the interval from the call to the constructor new `Worker(path)` until the worker posts its first readiness message.

Every result is recorded as JSON inside the directory `benchmarks/results/`, which permits regeneration of the article's tables from versioned artefacts.

5. Results of developing and empirically evaluating the formal model

5.1. Cost of error representations

The taxonomy of error representations developed under Objective 1 in Section 4.2 is empirically characterized here as part of Objective 3. The measured cost of error creation and escalation on Node.js 22.22.2 is given in Table 5.

Table 5. Cost of error creation and escalation (Node.js 22.22.2, V8 12.4, macOS 26.3, Apple M1 Pro)

Operation	Iterations	Measured cost
`new Error()` with stack capture	5 000 000	1.90 μs
`throw + catch` (synchronous)	5 000 000	2.01 μs
`Promise.reject + .catch`	2 000 000	1.99 μs
`Result.error({ code })`	20 000 000	7.1 ns
Typed code `{ code, data }`	20 000 000	5.1 ns

The data of Table 5 confirm hypothesis H1: creation of an exception is on average 270–400 times slower than creation of a `Result` value or of a typed code object. The principal cause is stack-trace capture inside the constructor of the `Error` class, which requires traversal of execution frames and string formatting of the resulting trace. On hot paths with expected failure modes such as input validation or configuration parsing, value-typed error representation removes this overhead entirely.

5.2. Overhead of recovery strategies

Objective 3 covers the empirical evaluation of the model components on the Node.js platform. The measured overhead of recovery strategies on successful and failure paths is given in Table 6.

The data of Table 6 confirm hypothesis H4: the median worker-restart latency through `worker_threads` is 19.7 ms, which is acceptable for applications in which workers perform batch computations lasting seconds or longer. The four strategies listed in hypothesis H2 – *retry with exponential back-off*, *Circuit Breaker*, *Saga* compensation, and supervision through `worker_threads` – are realized in the harness, and their happy-path overhead is reported in Table 6: 105 ns for retry (attempt 1 succeeds), 97 ns for the breaker in the closed state, 553 ns for a 3-step *Saga*, and 19.7 ms for a worker restart, relative to 46 ns for a direct `await`.

An unexpected outcome is the cost of fast failure of the breaker in the `open` state, which reaches 2.65 μs and is an order of magnitude higher than the breaker bookkeeping itself. The cause is the statement `throw new Error('Circuit open')` in the present implementation; the cost of this step

Table 6. Overhead of recovery strategies (Node.js 22.22.2, V8 12.4, macOS 26.3, Apple M1 Pro)

Strategy	Iterations	Happy path	Failure path
Direct `await` (no recovery)	2 000 000	46 ns	–
Retry, attempt 1 succeeds	2 000 000	105 ns	–
Retry, 3 attempts with 100 ms back-off, final failure	200	–	302 ms (p95 305 ms)
Breaker, state `closed` (nominal)	2 000 000	97 ns	–
Breaker, state `open` (fast fail)	2 000 000	–	2.65 μs
Saga, 3 steps, full success	1 000 000	553 ns	–
Saga, 3 steps, 3rd step fails, 2 compensations	1 000 000	–	3.16 μs
Worker restart (new `Worker` until ready event)	300	–	19.7 ms (median), 20.7 ms (p95)

is dominated entirely by the cost of exception construction shown in Table 5. In production systems, this observation motivates the use of a sentinel of the form `Result.error` on the fast-fail layer instead of a thrown exception, which reduces the overhead to approximately 200 ns.

5.3. Decomposition of exception cost

To explain why exception creation is two orders of magnitude more expensive than value-typed alternatives, two further measurements were performed. The first measurement varies the depth of the call stack at the point where new `Error()` is constructed. The second measurement varies the constructor used and the configuration of stack-trace capture. The dependence of new `Error()` cost on call-stack depth is given in Table 7.

Table 7. Cost of `new Error()` as a function of call-stack depth (Node.js 22.22.2, V8 12.4, macOS 26.3, Apple M1 Pro, 1 000 000 iterations)

Call-stack depth	Measured cost
1	1.77 μs
10	3.58 μs
50	3.79 μs
100	4.22 μs

The data of Table 7 demonstrate that the cost of `new Error()` grows from 1.77 μs at depth 1 to 3.58 μs at depth 10, and then saturates. The saturation point coincides with the default value of `Error.stackTraceLimit`, which is 10 in Node.js. Once the stack-trace capture has filled its budget of frames, additional call-stack depth does not contribute appreciably to the cost. This observation explains the long-running engineering folklore that "deeply nested code throws expensively" is correct only up to the stack-trace limit.

The dependence of the cost on the type of the thrown object and on the stack-trace configuration is given in Table 8.

The data of Table 8 establish three results. First, the built-in subclasses `TypeError` and `RangeError` carry no measurable penalty relative to the base `Error` class, so the choice of subclass is a stylistic decision rather than a performance decision. Second, a user-defined subclass that performs additional property assignment in its constructor is approximately 30% more expensive than the base class. Third, and most importantly, setting `Error.stackTraceLimit = 0` reduces the cost of `new Error()` from 1.34 μs to 0.21 μs, a reduction of approximately 6.4 times. Combined with the data of Table 7, this finding identifies stack-trace capture as the dominant cost component of exception construction. In

Table 8. Cost of `Error` subclasses and the effect of `Error.stackTraceLimit` (Node.js 22.22.2, V8 12.4, macOS 26.3, Apple M1 Pro, 5 000 000 iterations per row)

Constructor	Measured cost
`new Error("msg")`	1.34 μs
`new TypeError("msg")`	1.34 μs
`new RangeError("msg")`	1.34 μs
`new DomainError("msg", "CODE")` (custom subclass)	1.74 μs
`new Error("msg")` with `Error.stackTraceLimit = 0`	0.21 μs

production code on hot paths where stack traces are not consumed, lowering `Error.stackTraceLimit` is therefore a low-risk optimization.

5.4. Cross-boundary serialization cost

The qualitative characterization of cross-boundary fidelity in Table 2 is refined by measuring the cost of two standard serialization primitives applied to `Error` instances and to typed code objects. The results are given in Table 9.

Table 9. Cost of cross-boundary serialization by error representation (Node.js 22.22.2, V8 12.4, macOS 26.3, Apple M1 Pro)

Operation	Iterations	Measured cost
`structuredClone(error)` (`Error` instance)	500 000	787 ns
`structuredClone({ code, data })` (typed code)	5 000 000	677 ns
`structuredClone(string)` (short string code)	10 000 000	278 ns
`structuredClone` (nested code, 3 levels)	2 000 000	1.45 μs
`JSON.stringify(error)` (`Error` instance)	5 000 000	47 ns
`JSON.stringify({ code, data })` (typed code)	5 000 000	99 ns

The data of Table 9 yield two findings that strengthen the recommendations of Section 4.4. First, `structuredClone`, which is the primitive used by `MessagePort` to transfer values between workers, applies a fixed overhead of approximately 700 ns to both `Error` instances and typed code objects. The cost differential is small. The fidelity differential, however, is large: `structuredClone` of an `Error` preserves only the `message` and the `stack`, while own enumerable properties are dropped, as documented in Section 4.4. A typed code object, by contrast, is preserved without loss. Second, `JSON.stringify` of an `Error` instance returns the string `{}`, since the `Error` class declares no own enumerable properties. The 47 ns cost reported in Table 9 is therefore a measurement of producing a useless artifact. A typed code object serializes in 99 ns and preserves every field. The combination of comparable serialization cost with strictly better fidelity supports the assignment of typed codes to inter-process and distributed tiers in Section 4.6.

5.5. Integration with the architecture

The formal model is integrated with the Imperative Shell / Multi-Paradigm Core pattern in Section 4.6 and embodied in Table 4. The four-tier error architecture is summarized schematically below:

```
Tier 4: Distributed   - timeout, retry, breaker, Saga, error codes
Tier 3: Inter-process - supervisor, worker error events, typed codes
```

```
Tier 2: Intra-process - try / catch, unhandledRejection, EventEmitter errors
Tier 1: Core          - Result monad, total functions, no error escalation
```

The distribution of representations across tiers is supported by the data of Tables 5–6. Total functions in the core eliminate error escalation entirely, so the core pays no cost for exception construction at all. For higher tiers, where input-output and cross-boundary interaction are unavoidable, the cost of exceptions remains acceptable because it is amortised by the latency of the input-output operation itself. The resulting distribution provides quantitative justification for the recommendations of Section 4.6.

5.6. Quantitative recommendations derived from the model evaluation

The empirical evaluation of the model yields quantitative recommendations that follow directly from Tables 5–9 and constitute the practical contribution of this study. The guidelines are formulated as five rules. The corresponding decision matrix is given in Table 10.

Rule 1. Default to value-typed error representations within the pure core of an application. The data in Tables 5 and 8 establish that exception construction costs approximately 1.9 μs while a `Result` value costs approximately 7 ns and a typed code object costs approximately 5 ns. The 270-fold cost gap dominates any cost associated with the additional allocation paid by `Result`, since the additional allocation is implicit in object construction in either case. The rule applies unconditionally inside the core.

Rule 2. Reserve thrown exceptions for boundary code where the call-site stack trace is consumed. The decomposition in Table 8 attributes more than 80% of exception cost to stack-trace capture. On the imperative shell, the stack trace is typically logged or attached to a fault report and is genuinely useful for debugging, which justifies the cost. On hot inner-loop paths the trace is rarely consumed, so the cost is wasted.

Rule 3. On boundaries where failure is expected and frequent, prefer sentinels of type `Result.error` over thrown exceptions for fast-fail signalling. The breaker fast-fail measurement in Table 6 (2.65 μs) is dominated by the construction of `new Error('Circuit open')`. Replacing the thrown exception with a sentinel value reduces the fast-fail cost to approximately 200 ns, a 13-fold improvement at no loss of behavioural information.

Rule 4. Apply `Error.stackTraceLimit = 0` on the most performance-sensitive paths where exception construction cannot be avoided. Table 8 reports that this setting reduces the cost of new `Error()` from 1.34 μs to 0.21 μs, a 6.4-fold improvement, while leaving the `Error.name` and `Error.message` fields intact. The setting is a process-global flag, so the decision must be coordinated across the application. The trade-off is the loss of stack-trace information for debugging, which justifies the rule only when stack traces are not consumed.

Rule 5. Choose typed codes rather than `Error` instances at every serialization boundary. Table 9 establishes that `structuredClone` of an `Error` and of a typed code object cost approximately 787 ns and 677 ns respectively, a difference of less than 16%. The fidelity differential, however, is large: `Error` instances lose every own enumerable property, and `JSON.stringify` of an `Error` returns the empty object `{}`. Typed codes preserve every field at marginal additional cost.

Table 10. Decision matrix for the choice of error representation by execution context

Execution context	Recommended representation	Source rule	Source table
Pure functions inside the core	*Result monad*	Rule 1	Tables 5 and 8
Validation and parsing on the imperative shell	*Result monad* or typed code	Rule 1, Rule 2	Tables 5 and 8
Input-output boundary handlers	Thrown `Error` (stack trace consumed)	Rule 2	Tables 5 and 8

Table 10. Decision matrix for the choice of error representation by execution context (continued)

Execution context	Recommended representation	Source rule	Source table
Fast-fail on tripped `CircuitBreaker`	`Result.error` sentinel	Rule 3	Table 6
Hot path with frequent failures and unused stacks	Thrown `Error` with `Error.stackTraceLimit = 0`	Rule 4	Table 8
Worker-to-worker `MessagePort` payload	Typed code object	Rule 5	Table 9
Distributed RPC response body	Typed code object plus HTTP status	Rule 5	Tables 2 and 9

The recommendations in Table 10 form the operational complement to the architectural recommendations of Section 4.6 and are quantitatively grounded in the measurements of Section 5.1–5.4.

6. Discussion of the results of developing and evaluating the formal model

The obtained results address several questions that have not been settled in earlier work. First, the hypothesis of a two-order-of-magnitude gap between the cost of exceptions and the cost of error values [4, 5] is confirmed quantitatively. The measurements in Table 5 show that exceptions are 270–400 times slower because of stack-trace capture, which is explained by the internal protocol of V8 for constructing `Error` objects. The further decomposition in Tables 7 and 8 attributes more than 80% of that cost to stack-trace capture. Setting `Error.stackTraceLimit = 0` collapses the cost of new `Error()` from 1.34 μs to 0.21 μs, while the choice of subclass contributes no measurable penalty. This result supports the use of the *Result monad* and of typed error codes on hot application paths, and identifies `Error.stackTraceLimit` as a low-risk knob for hot paths where stack traces are not consumed.

Second, the results on recovery strategies show that the overhead on the happy path for *retry*, *breaker*, and *Saga* ranges from tens to hundreds of nanoseconds. This finding implies that the typical warning against the "excessive" use of such wrappers on hot paths is not justified for runtimes with a modern just-in-time compiler. In the context of the patterns introduced in [10, 11], the present measurements move the *discussion* from a qualitative to a quantitative footing.

Third, the observed "fast-fail paradox" is a consequence of using the `throw` statement to signal failure. The latency of the breaker fast-fail is an order of magnitude higher than the breaker bookkeeping itself: 2.65 μs versus 97 ns. This finding reinforces the recommendation of Section 5.2: on boundaries where failures are frequent and expected, value-typed error representations deliver substantial savings.

Fourth, the median worker-restart latency of 19.7 ms confirms the practicality of the "let it crash" architectural pattern [12] in the Node.js environment. The study [5] argues for supervision analytically; the present measurement provides a quantitative budget within which supervision remains viable. For requests with an expected latency below two milliseconds, worker restart cannot be the primary strategy. For batch tasks lasting seconds or longer, supervision is the natural choice.

The interpretation of the obtained results agrees with the conclusions of the works [13–15] on distributed systems: the further an error crosses boundaries, the less practical it is to rely on a full-fidelity `Error` object. The measurements in Table 9 quantify the fidelity gap: although `structuredClone` applies a near-identical cost of approximately 700 ns to `Error` instances and to typed code objects, the resulting payloads differ sharply in faithfulness. `Error` instances lose their own enumerable properties, and `JSON.stringify` of an `Error` returns the empty object `{}`, which

makes JSON unusable as a transport for exception data. The four-tier architecture proposed in this study (Section 4.6) formalizes this intuition through the choice of error representation according to the fidelity and the delivery guarantees of each boundary.

The practical value of the obtained results consists in the direct application of the recommendations to the design of fault-tolerant Node.js systems. As supplementary material, the study provides a reproducible benchmark harness that allows other researchers to obtain comparable measurements on target hardware.

On the basis of the taxonomy, a formal model of error escalation and recovery has been developed. The model comprises three components. First, escalation semantics across event-loop phases and across worker-thread, child-process, and network boundaries, including Propositions 1 (*Promise rejection visibility*) and Proposition 2 (*EventEmitter error invariant*) on `Promise` rejection visibility and the *EventEmitter* error invariant. Second, contracts for four principal recovery strategies – *retry with exponential back-off*, *Circuit Breaker*, *Saga* compensation, and supervision trees adapted to Node.js worker threads – with idempotency and placement preconditions made explicit. Third, placement rules that extend the Imperative Shell/Multi-Paradigm Core pattern from a separation of side effects to a separation of error responsibilities, formulated as hypothesis H3. Unlike the asynchronous-execution semantics of [7] and the `Promise` typing of [9], which stop at the language boundary, the proposed model formalizes escalation across four boundary types and treats cross-boundary fidelity as a property of the escalation rule [10–12], who each formalize one recovery strategy in isolation, the proposed model places the four strategies inside the taxonomy-and-IS/MPC frame. Design-by-contract [16] is the closest source of formal placement reasoning but predates asynchronous execution and does not address error representation. The scientific novelty of the study consists in unifying four elements within this frame: the representation taxonomy, escalation across four boundary types, contracts for four recovery strategies, and the error-aware IS/MPC placement rule. The model was developed in order to make the per-layer assignment of representations and recovery strategies a predictable, measurable design decision. The obtained effect is the architectural frame within which Objective 3 evaluates representation cost (Tables 5, 7 and 8), cross-boundary fidelity (Table 9), and recovery-strategy overhead (Table 6).

An empirical evaluation of the model has been performed on the Node.js platform. A reproducible benchmark harness, provided as supplementary material, measures four model-aligned axes: representation cost (Tables 5, 7 and 8), cross-boundary serialization fidelity (Table 9), recovery-strategy overhead (Table 6), and a decomposition that attributes exception cost to stack-trace capture (Tables 7 and 8). Unlike prior studies of asynchronous JavaScript, which target language semantics [7, 9] or static analysis [17], the harness empirically characterizes the cost of error-handling design choices and the fidelity of error data across architectural boundaries. The evaluation was performed in order to convert the model components from design proposals into measurable engineering claims. The obtained effect is the confirmation of hypotheses H1 and H4, support for H3 through the tier-distribution analysis of Section 5.5, the realization of the four recovery strategies of hypothesis H2 with happy-path overhead reported in Table 6, and the decision matrix of Table 10, which is the operational basis of the recommendations of Section 5.6 and the practical-value statement in Section 6.

Limitations of the study. The analysis is focused on the Node.js/V8 platform. For browser environments, the lifecycle management of `ServiceWorker` and the semantics of `unhandledRejection` require separate treatment. The experimental measurements were performed on a single hardware platform (Apple M1 Pro). Cross-platform comparison remains a task for future work. The benchmarks evaluate the cost of representations and strategies under micro-synthetic conditions. The impact of realistic workloads on breaker state and supervisor behaviour requires field studies.

Directions for future research include the following.

1. Formal verification of supervisor restart bounds via the temporal logic TLA+.

2. Integration of breaker state with distributed health monitoring based on CRDTs.
3. Empirical study of error-escalation patterns in production Node.js systems.
4. Extension of the TypeScript type system for statically checked asynchronous error channels.
5. Automatic placement of error boundaries on the basis of static analysis of `await` dependency graphs.

Conclusion

A taxonomy of error representations for event-loop runtimes has been developed. The taxonomy formalizes each representation as a triple of value domain, signalling mechanism, and escalation rule, and characterizes four representations – thrown exceptions, *error-first callbacks*, the *Result monad*, and typed error codes – along composability, signature visibility, cross-boundary fidelity, control-flow interruption, stack-trace debuggability, and paradigm affinity. It unifies four previously isolated approaches to error representation within a single contract format and introduces cross-boundary fidelity as a first-class axis of architectural analysis, an axis absent from prior frameworks that treat exceptions or value-based representations in isolation. As a result, the choice of representation at each architectural layer becomes a principled, contract-oriented design decision, providing a uniform basis for the boundary-specific escalation semantics of Objective 2 and the cost measurements of Objective 3.

On the basis of the developed taxonomy, a formal model of error escalation and recovery in layered asynchronous architectures with an event loop has been developed. The model integrates escalation semantics across execution boundaries, contracts of four recovery strategies, namely *retry with exponential back-off*, *Circuit Breaker*, *Saga* compensation, and supervision tree, and placement rules within the Imperative Shell/Multi-Paradigm Core architecture. Unlike existing approaches that separately consider asynchronous execution, error representation, or individual recovery strategies, the proposed model treats errors as cross-layer architectural events and formalizes the choice of recovery strategy as a contract-oriented design decision. The scientific novelty consists in combining representation taxonomy, boundary-aware escalation, and unified recovery-strategy contracts within a single architectural model. The developed model made it possible to form a formal architectural framework for systematic analysis of representation cost, cross-boundary fidelity, and recovery-strategy overhead, which supports predictable selection of error representations and recovery mechanisms at different architectural layers.

The proposed model was empirically evaluated on the Node.js platform. The results show that the model reduces error-handling overhead by using value-typed representations on hot paths, improves cross-boundary fidelity through typed error codes, and enables predictable use of recovery strategies with measurable overhead. These results confirm the practical applicability of the model and its suitability for engineering use in fault-tolerant Node.js systems.

Acknowledgements

Declaration on the use of Artificial Intelligence. The authors used a generative AI tool during the preparation of this manuscript solely for formatting of the document. All AI-assisted formatting was reviewed by the authors. The authors take full responsibility for the accuracy, integrity, and originality of the article.

References

[1] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *Proc. 11th USENIX Symp. Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, USA, Oct. 2014, pp. 249–265, accessed: May 17, 2026. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

[2] Information Technology Intelligence Consulting, "ITIC 2024 hourly cost of downtime report," Tech. Rep., 2024. https://itic-corp.com/itic-2024-hourly-cost-of-downtime-report/, accessed: May 17, 2026.

[3] J. B. Goodenough, “Exception handling: Issues and a proposed notation,” *Commun. ACM*, vol. 18, no. 12, pp. 683–696, Dec. 1975. https://doi.org/10.1145/361227.361230.
[4] P. Wadler, *Monads for Functional Programming*, ser. Lecture Notes in Computer Science. Berlin: Springer, 1995. https://doi.org/10.1007/3-540-59451-5_2, vol. 925, pp. 24–52.
[5] C. Hewitt, “Actor model of computation: Scalable robust information systems,” 2010. [Online]. Available: https://doi.org/10.48550/arXiv.1008.1459
[6] B. Liskov and L. Shrira, “Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems,” in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 1988, pp. 260–267. [Online]. Available: https://doi.org/10.1145/53990.54016
[7] M. C. Loring, M. Marcotte, and D. Kapur, “Semantics of asynchronous JavaScript,” in *Proc. 13th ACM SIGPLAN Int. Symp. Dynamic Languages (DLS)*, 2017, pp. 51–62. [Online]. Available: https://doi.org/10.1145/3133841.3133846
[8] WHATWG, “HTML living standard,” 2026. https://html.spec.whatwg.org/multipage/webappapis.html#event-loops, accessed: May 15, 2026.
[9] M. Madsen, O. Lhoták, and F. Tip, “A model for reasoning about JavaScript promises,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 1–24, Oct. 2017. https://doi.org/10.1145/3133909, art. 86.
[10] M. Nygard, *Release It! Design and Deploy Production-Ready Software*, 2nd ed. Raleigh, NC, USA: Pragmatic Bookshelf, 2018.
[11] H. Garcia-Molina and K. Salem, “Sagas,” in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 1987, pp. 249–259. [Online]. Available: https://doi.org/10.1145/38713.38742
[12] J. Armstrong, “Making reliable distributed systems in the presence of software errors,” Ph.D. dissertation, Royal Inst. Technol., Stockholm, Sweden, 2003. https://erlang.org/download/armstrong_thesis_2003.pdf, accessed: May 15, 2026.
[13] M. Kleppmann and A. R. Beresford, “A conflict-free replicated JSON datatype,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 10, pp. 2733–2746, Oct. 2017. https://doi.org/10.1109/TPDS.2017.2697382.
[14] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “Conflict-free replicated data types,” in *Proc. 13th Int. Symp. Stabilization, Safety, and Security of Distributed Systems (SSS)*, ser. Lecture Notes in Computer Science, vol. 6976. Berlin: Springer, 2011, pp. 386–400. [Online]. Available: https://doi.org/10.1007/978-3-642-24550-3_29
[15] P. S. Almeida, A. Shoker, and C. Baquero, “Delta state replicated data types,” *J. Parallel Distrib. Comput.*, vol. 111, pp. 162–173, Jan. 2018. https://doi.org/10.1016/j.jpdc.2017.08.003.
[16] B. Meyer, “Applying ‘design by contract’,” *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992. https://doi.org/10.1109/2.161279.
[17] M. Madsen, F. Tip, and O. Lhoták, “Static analysis of event-driven Node.js JavaScript applications,” in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015, pp. 505–519. [Online]. Available: https://doi.org/10.1145/2814270.2814272

УДК 004.41:004.272.43

ФОРМАЛЬНА МОДЕЛЬ ЕСКАЛАЦІЇ ПОМИЛОК ТА ВІДНОВЛЕННЯ В БАГАТОШАРОВИХ АСИНХРОННИХ АРХІТЕКТУРАХ ІЗ ЦИКЛОМ ПОДІЙ

Дмитро Нечай
https://orcid.org/0000-0002-0696-9985

Тимур Шемсединов
https://orcid.org/0000-0001-5958-4731

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського», Київ, Україна

Отримано: 27.04.2026р. / Прийнято: 11.05.2026р. / Опубліковано: 27.05.2026р.

Багатошарові асинхронні архітектури з циклом подій поєднують планування всередині процесу, багатопотоковому середовищі, ізоляцію дочірніх процесів та мережеву взаємодію. Їхня надійність залежить від того, як виключення представлені, ескалуються, відновлюються та перетворюються на архітектурних межах. Об'єктом цього дослідження є ескалація та відновлення після помилок у таких архітектурах. Метою є підвищення надійності багатошарових асинхронних систем шляхом розробки та емпіричної оцінки формальної моделі, яка пов'язує представлення помилок, межі ескалації, стратегії відновлення та накладні витрати в рамках шаблону *Imperative Shell / Multi-Paradigm Core*. У дослідженні використовується формальне порівняння контрактів представлення помилок, аналіз *event-loop*, *worker-thread*, *child-process*, та мережевого зв'язку, архітектурне моделювання та відтворюваний мікробенчмаркінг на платформі *Node.js*. Запропонована модель включає таксономію чотирьох представлень помилок: винятки, зворотні виклики *error-first*, монада *Result* та коди помилок. Вона також визначає семантику ескалації для фаз циклу подій та меж процесу, потоків та мережі. Модель призначає обчислення багатопарадигмальному ядру, а побічні ефекти відновлення, логування, повтори, *circuit-breaker*, компенсацію, контроль та перетворення протоколу – імперативній оболонці. Емпірична оцінка показує, що викидані помилки в сотні разів дорожчі, ніж альтернативи на основі типів значень, що обгортки повторних спроб та *circuit-breaker* додають лише десятки наносекунд до успішного шляху, а затримка перезапуску потоків становить близько 20 мс. Наукова новизна полягає в поєднанні таксономії представлення, гранично-чутливої семантики ескалації, поєднання стратегії відновлення та вимірюваних накладних витрат у формальну модель. Отримані рекомендації допомагають проектуванню відмовостійких систем на базі *Node.js*.

Ключові слова: ескалація помилок, асинхронна архітектура, цикл подій, представлення помилок, стратегія відновлення, монада Result, дерево супервізії, відмовостійкість, JavaScript, Node.js.