# Design for a New Error Reporting System for Workflow Launcher (WFL)

## 1. Executive Summary

This report outlines the design for a new, comprehensive error reporting system for the Workflow Launcher (WFL). The current error reporting mechanisms within WFL, while possessing foundational elements, exhibit limitations that impede developer productivity, slow down issue resolution, and potentially impact overall system reliability. The proposed system is designed from the ground up, drawing insights from established best practices in error reporting [1] and tailored to WFL's specific architectural and operational context.

The strategic imperative for this new system is underscored by WFL's ambition to support sophisticated operational data acquisition and optimization, akin to a "smart factory" environment.[2] Achieving such goals necessitates robust, real-time visibility into system behavior, particularly its failure modes. Inadequate error handling can lead to developer frustration, increased support overhead, and application instability, all of which are antithetical to efficient optimization.[1] The current state, characterized by a somewhat implicit and under-documented error handling approach [3], represents not merely technical debt but a direct constraint on realizing WFL's full potential for enhanced productivity and process effectiveness.[2]

The new error reporting system is founded on core architectural pillars:

- **Unified Error Handling:** Standardized error definitions, propagation, and handling across all WFL components and execution stages.
- **Context-Aware Logging:** Richly detailed log events that provide comprehensive diagnostic information.
- **Intelligent Monitoring and Alerting:** Proactive detection, grouping, and notification of errors, minimizing noise and prioritizing actionable issues.
- **Security by Design:** Ensuring error communication does not compromise sensitive data or system integrity.

This system is designed to align seamlessly with WFL fundamentals, including its Clojure-based architecture, staged workload model (source, executor, sink), and cloud-native deployment.[3] By prioritizing developer experience (DX), the new system aims to transform error reports from cryptic puzzles into valuable diagnostic tools, thereby enhancing developer productivity, reducing Mean Time To Resolution (MTTR), improving system reliability, and providing superior operational visibility. Ultimately, this redesigned error reporting system will serve as a strategic enabler for WFL's broader objectives of operational excellence and continuous improvement.

## 2. Current State Analysis: Error Reporting in WFL

A thorough understanding of WFL's current error reporting landscape is essential before designing a new system. This analysis examines WFL's architecture, existing logging and monitoring capabilities, and implicit error handling practices to identify strengths, weaknesses, and critical gaps.

## 2.1. WFL Architecture and Key Execution Phases

Workflow Launcher (WFL) is a Clojure-based system [3] designed to manage and execute scientific workflows. Its architecture is characterized by a staged workload model, which typically involves:

1. **Source Stage:** Fetching data and metadata required for a workflow. This may involve interacting with services like Google Cloud Storage (GCS) or internal data repositories like DataRepo.[3]
2. **Executor Stage:** Submitting the workflow to an execution engine, predominantly Cromwell, and monitoring its progress. This stage involves interaction with the Cromwell server and potentially other cloud resources.[3]
3. **Sink Stage:** Delivering the results of the completed workflow to a specified output location, such as GCS, and potentially updating databases or triggering downstream processes via mechanisms like Java Message Service (JMS).[3]

WFL is designed for cloud-native deployment, primarily on Kubernetes clusters, and leverages Google Cloud Platform (GCP) services extensively, including GCS and the Google Cloud SDK.[3] Key Clojure modules within WFL manage interactions with these components, such as cromwell.clj, datarepo.clj, gcs.clj, and jms.clj.[3]

The distributed, multi-stage nature of WFL, combined with its reliance on several external systems, inherently creates complex error propagation pathways. A failure originating in an early stage, such as data fetching from a source, or an issue within an external service like Cromwell, can cascade through the system. Without robust correlation mechanisms, these errors might manifest obscurely in a later stage, such as the sink, making root cause analysis exceedingly challenging. For instance, an error reported during result delivery could stem from incorrect input parameters defined at the source stage or a transient failure within the Cromwell execution environment. This potential for diagnostic opacity underscores the necessity for an integrated error reporting system capable of tracing issues across these distributed boundaries, a common requirement in complex systems to avoid prolonged debugging cycles.[1]

## 2.2. Evaluation of Existing Logging Mechanisms

WFL currently employs a logging system that utilizes macros to generate JSON-formatted log records. These logs are compatible with Google Stackdriver severity levels (e.g., INFO, ERROR), and WFL provides an API endpoint (/logging_level) to dynamically adjust the logging verbosity.[3]

**Strengths:**

- **Structured Logging:** The use of JSON for log entries is a significant strength, as structured logs are machine-parsable, enhancing searchability, filtering, and analysis.[1]

- **Centralization via Stackdriver:** Sending logs to Google Stackdriver provides a centralized platform for log aggregation and storage, which is crucial for distributed systems.[1]
- **Dynamic Log Levels:** The ability to change log levels at runtime is beneficial for targeted debugging without requiring application restarts.[3]
- **Namespace-Qualified Keys:** The translation of specific Clojure keywords (e.g., ::log/spanId) to Stackdriver-recognized keys (e.g., logging.googleapis.com/spanId) shows an attempt to align with cloud logging conventions.[3]

**Weaknesses:**

- **Limited Contextual Depth for Errors:** While logs are structured, the documented examples primarily show basic information like severity and a span ID.[3] There is no explicit mention of a standardized, rich schema for capturing WFL-specific contextual information critical for error diagnosis, such as workflow identifiers, granular stage information (source, executor, sink), failing input parameters (sanitized), or user identifiers. Best practices emphasize the need for comprehensive context like stack traces, request correlation IDs, user/session data, input values, and environment details to effectively diagnose errors.[1]
- **Reliance on Basic Search Tools:** The documentation suggests using command-line tools like grep and jq for local log searching.[3] While useful, this implies that the logged data may not be consistently structured with rich, error-specific queryable fields that would facilitate more advanced or targeted investigations directly within Stackdriver or other analysis tools.
- **Potential for Insufficient Error Narratives:** The existing logging might adequately record that an event occurred but may not consistently provide enough information to understand *why* an error resulted from it or its precise impact domain (e.g., which specific samples in a batch workload were affected, or if the failure impacted a critical business process). If such detailed error-specific data isn't systematically captured, developers will face difficulties in reproducing errors or assessing their full scope, leading to increased Mean Time To Resolution (MTTR).[1] The current system appears more geared towards general operational logging rather than specialized, efficient error diagnostics.

## 2.3. Assessment of Current Monitoring and Alerting Capabilities

WFL's monitoring approach involves sending stdout and stderr streams to Google Logging (Stackdriver). Within Stackdriver, metrics can be created from log queries, and alerts can be configured based on these metrics. Notifications for these alerts can be routed to various channels, including Slack and email.[3]

**Strengths:**

- **Leverages GCP Infrastructure:** Utilizing Google Cloud's native monitoring and alerting capabilities provides a scalable and integrated foundation, reducing the need to manage separate infrastructure for these functions.[3]

**Weaknesses:**

- **Reactive Alerting:** The described alerting mechanism is primarily reactive, triggered by predefined queries against logs (e.g., resource.labels.container_name="workflow-launcher-api" severity>=ERROR).[3] This approach may not detect nuanced error conditions or provide early warnings for degrading performance that could lead to errors.
- **Lack of Intelligent Error Processing:** The system does not appear to incorporate intelligent error detection features such as anomaly detection, automatic grouping of similar error instances into single "issues," or impact assessment beyond simple error counts derived from log queries. Dedicated error monitoring tools often provide these capabilities, which are crucial for reducing alert noise and helping developers prioritize effectively.[1]
- **Potential for Alert Fatigue:** Alerting on generic conditions like any log entry with severity>=ERROR can lead to "alert fatigue," where developers become desensitized to notifications due to a high volume of non-critical or redundant alerts.[1] This desensitization can result in genuinely critical alerts being overlooked.
- **Developer Burden for Configuration:** The responsibility for defining meaningful metrics and alert policies falls heavily on developers or operators. Without specific, system-wide guidance or more sophisticated tooling for WFL error patterns, the application of these alerts might be inconsistent or suboptimal. The example query provided [3] is very broad and likely insufficient for fine-grained, actionable alerting.

The current monitoring setup, while functional for basic threshold-based alerting, likely suffers from the common pitfall of generating excessive noise or missing subtle but significant error patterns. Its reliance on manually configured, simple log-based metrics means it doesn't inherently offer the "intelligent error grouping" or "rich contextual data capture" that modern error monitoring solutions advocate.[1] This suggests that developers might spend considerable time sifting through generic alerts or manually correlating information, rather than receiving focused, context-rich notifications about prioritized issues, thereby impacting productivity and potentially delaying the response to critical system failures.

## 2.4. Deep Dive into WFL's Implicit and Explicit Error Handling

The official WFL documentation, as reviewed, lacks a dedicated section detailing its error handling strategies.[3] The primary explicit mention of error handling is the validation of command-line arguments within the main.clj entry point.[3] Beyond this initial validation, error management within WFL appears to be largely implicit, likely relying on:

- Standard Clojure try-catch blocks implemented by developers within individual modules.
- The propagation of Java exceptions, given that Clojure operates on the Java Virtual Machine (JVM) and interacts with Java libraries.[4]
- The inherent error reporting behaviors of integrated external services, such as Cromwell, GCS, or JMS.

In the absence of a formalized, documented system-wide error handling strategy, it is probable that error management practices are ad-hoc, varying significantly between different components of WFL and according to individual developer preferences or habits. This

variability can lead to several challenges. For instance, while Clojure provides powerful mechanisms for conveying rich error information, such as attaching arbitrary data maps to exceptions using ex-data [4], the consistent and effective utilization of such features requires deliberate design and established conventions across the codebase. Without these, the error information captured may be inconsistent in structure and content.
This ad-hoc approach significantly increases the risk of common error handling pitfalls, such as:

- **Overly broad exception catching:** Using generic catch Exception e blocks that might inadvertently mask critical, unrecoverable system errors.
- **Swallowing errors:** Empty catch blocks or inadequate logging of caught exceptions can hide the root cause of problems, leading to silent failures where the system continues to operate in an unknown or erroneous state.[1]
- **Loss of context:** Failure to properly wrap exceptions or propagate crucial contextual information when errors are caught and re-thrown. This makes subsequent debugging exponentially more difficult, as developers are left dealing only with symptoms rather than root causes.

Consequently, developers tasked with diagnosing and resolving issues in WFL likely encounter a heterogeneous landscape of error signals. The lack of standardized error types, inconsistent payload structures (if ex-data is used sporadically or without convention), and varying levels of detail in error messages can significantly increase the cognitive load and time required for debugging. This directly undermines developer experience and operational efficiency.[1]

## 2.5. Summary of Strengths, Weaknesses, and Critical Gaps in the Current System

The current error reporting mechanisms in WFL present a mixed picture:
**Strengths:**

- **Foundation in Structured Logging:** The adoption of JSON for log output provides a good base for machine readability and integration with log management systems.[3]
- **Centralized Logging via Stackdriver:** Utilizing Google Stackdriver for log aggregation offers a scalable, centralized platform for collecting log data from WFL's distributed components.[3]
- **Basic Monitoring and Alerting Framework:** The ability to create metrics from logs and configure alerts in Stackdriver provides a rudimentary capability for operational monitoring.[3]
- **Dynamic Log Level Adjustment:** The API for changing log verbosity at runtime is a useful feature for debugging specific issues.[3]

**Weaknesses:**

- **Lack of Comprehensive Error-Specific Context in Logs:** Logs often lack the detailed, WFL-specific contextual information necessary for rapid and effective error diagnosis.
- **Reactive and Potentially Noisy Alerting:** Alerting is primarily based on simple log queries (e.g., severity levels), leading to reactive responses and a high potential for alert fatigue.

- **Absence of a Unified Error Handling Strategy:** There is no documented, system-wide approach to error definition, capture, and propagation, leading to inconsistent practices.
- **Inconsistent Error Information:** Error messages and payloads (where they exist) likely vary in structure and content across different parts of the application.
- **Difficulty Correlating Errors:** Tracing errors across WFL's distributed components (source, executor, sink) and its interactions with external services like Cromwell is likely challenging due to a lack of consistent correlation IDs and contextual linkage in error reports.

**Critical Gaps:**

- **No Proactive Error Detection or Intelligent Grouping:** The system lacks mechanisms for intelligent error aggregation, de-duplication, anomaly detection, or proactive identification of emerging issues.
- **Insufficient Detail for Rapid Root Cause Analysis:** Developers often do not have all the necessary information readily available to quickly understand the "where, what, when, and how" of an error.
- **Potential for Silent Failures:** The ad-hoc nature of error handling increases the risk of errors being caught and suppressed without adequate logging or notification.
- **Suboptimal Developer Experience:** The current system likely leads to increased debugging time and frustration for developers, falling short of the ideals of a developer-centric error reporting system.[1]

The cumulative effect of these weaknesses and gaps is that WFL's current error reporting system functions more as a passive, forensic tool—used primarily after an issue has become apparent—rather than an active, diagnostic, and preventative system. This passivity limits WFL's operational maturity and its capacity to reliably support the complex, high-volume workloads associated with its "smart factory" aspirations.[2] A system that merely records some data after failure, rather than actively guiding developers, intelligently grouping issues, and helping to prevent future occurrences, acts as a drag on efficiency and reliability.

The following table summarizes the current state and points towards the enhancements proposed by the new system design:

**Table 1: WFL Error Handling: Current State vs. Proposed Enhancements**

| Feature Area | Current State (Inferred/Documented) | Proposed Enhancements (New System) | Benefit of Enhancement |
|---|---|---|---|
| **Error Handling Strategy** | Ad-hoc, implicit; relies on individual developer practices and basic Clojure/Java mechanisms.[3] | Unified framework with standardized error codes, structured payloads (via ex-data), and defined handling per WFL stage.[4] | Consistency, predictability, easier debugging, clear ownership of errors, machine-readable error data. |
| **Log Content &** | Structured JSON, but | Comprehensive, | Rapid root cause |

| | | | |
|---|---|---|---|
| **Context** | potentially lacking deep WFL-specific error context (e.g., workflow ID, input data).[3] | standardized error log schema with rich WFL-specific context, correlation IDs, and stack traces.[1] | analysis, easier error reproduction, ability to trace errors across distributed components. |
| **Monitoring Approach** | Reactive; metrics derived from basic log queries in Stackdriver (e.g., severity>=ERROR).[3] | Proactive; intelligent error detection, aggregation, grouping of similar errors, anomaly detection, and impact analysis.[1] | Reduced alert noise, faster identification of unique/impactful bugs, early warning of systemic issues, focus on high-priority problems. |
| **Alerting Intelligence** | Basic threshold alerting; potential for alert fatigue.[1] | Actionable alerts based on new error types, error rate spikes, user impact; direct links to detailed error reports and logs.[1] | Fewer, more meaningful alerts; quicker access to diagnostic information; reduced MTTA. |
| **API Error Handling** | Undocumented; likely inconsistent if WFL exposes external APIs. | Standardized API error responses (e.g., RFC 9457), accurate HTTP status codes, secure error messages.[1] | Improved developer experience for API consumers, more robust integrations with client systems, enhanced security. |
| **Developer Experience** | Potentially frustrating due to vague errors, missing context, and inconsistent signals.[1] | Developer-centric design: clear messages, rich context, actionable guidance, easy-to-use instrumentation libraries.[1] | Reduced debugging time, lower developer frustration, increased productivity, faster onboarding to error handling practices. |
| **Security** | No explicit security measures documented for error reporting; potential for data leakage. | Security by design: sanitization of sensitive data in logs/messages, generic auth errors, controlled information disclosure.[1] | Protection against inadvertent sensitive data exposure through errors; reduced attack surface. |

# 3. Foundational Principles for the New WFL Error Reporting System

The design of the new error reporting system for WFL will be guided by a set of core

principles. These principles ensure that the system is not only technically sound but also aligns with WFL's operational goals, enhances developer productivity, and leverages the unique strengths of its underlying technology stack.

## 3.1. Aligning with WFL Fundamentals

The new error reporting system must be intrinsically woven into the fabric of WFL's architecture and operational philosophy. "WFL fundamentals" encompass several key aspects:

- **Staged Processing Model:** WFL's core operation involves distinct source, executor, and sink stages.[3] The error system must provide clear visibility into how errors originate and propagate across these stages. This implies robust mechanisms for error contextualization and correlation that can trace a single workload's journey and pinpoint failures accurately within this distributed flow.
- **Cloud-Native Architecture:** WFL's deployment on Kubernetes and its reliance on Google Cloud services [3] dictate that the error reporting system should be designed to integrate seamlessly with and leverage cloud infrastructure for scalability, resilience, and potentially cost-effectiveness. However, it should also maintain a degree of portability to adapt to future platform evolutions.
- **Process Safety and Future-Readiness:** WFL aims for "maximisation of process safety" and to be "future-ready due to the best connectivity".[6] An advanced error reporting system is a direct contributor to process safety by enabling rapid detection and remediation of issues that could compromise workflow integrity. Future-readiness implies that the error system itself must be adaptable.
- **Evolvability and Continuous Improvement:** The broader context of modern software development, as highlighted in discussions around software execution phases [7] and fundamental architectural principles [8], emphasizes iterative refinement and adaptation. "Fundamental things continue to change at a rapid pace in the software world," and architectures must be prepared to "question fundamental axioms on a regular basis".[8] Therefore, a static error reporting system will not suffice. The new system must be designed for evolvability, allowing new types of errors, additional contextual data points, and enhanced analytical capabilities to be incorporated smoothly as WFL itself evolves, adds new processing capabilities, integrates with new external services, or encounters different workload characteristics. This inherent adaptability is a crucial "fundamental" for the error reporting system, ensuring it remains a valuable asset rather than becoming legacy overhead.

## 3.2. Embracing Developer-Centric Error Reporting

The paramount principle guiding this redesign is the unwavering focus on Developer Experience (DX), as extensively detailed in "Optimizing Error Reporting for Enhanced Developer Experience".[1] Every design decision, from data capture to presentation, will be evaluated based on its utility to the WFL developers who will diagnose and resolve errors. This commitment translates into prioritizing the following characteristics [1]:

- **Clarity and Precision:** Error messages must be unambiguous, easily comprehensible,

and directly guide developers toward the root cause. Jargon should be relevant and understandable to the target developer audience.

- **Rich Contextual Information:** Reports must answer the "where, what, when, and how" of an error, providing comprehensive details like stack traces, request/correlation IDs, user/session data (if applicable), input values, and environment specifics.
- **Consistency and Standardization:** Error formats, codes, and terminology must be uniform across all WFL modules and services to ensure predictability and facilitate automated analysis.
- **Non-Intrusiveness and Performance:** The error reporting mechanism itself should have minimal performance impact on WFL during normal operation, with costs primarily incurred only when an error is actively being reported.
- **Actionability:** Error reports should not merely state problems but actively guide developers toward solutions, potentially suggesting fixes, linking to relevant documentation, or outlining next diagnostic steps.
- **Security-Consciousness:** All error communications must be carefully crafted to prevent the inadvertent exposure of sensitive data.

True developer-centricity, however, extends beyond the final error *report* to the *instrumentation* process itself.[1] The ease with which WFL developers can generate high-quality, contextualized errors from their Clojure code is critical. If the APIs, libraries, or conventions for signaling errors within WFL are cumbersome, difficult to use, or poorly documented, adoption of best practices will be low, thereby undermining the entire system's effectiveness by starving it of rich, informative error data. Therefore, the design must meticulously consider the "producer DX"—the experience of WFL developers instrumenting their code—just as much as the "consumer DX"—the experience of those diagnosing reported errors.

## 3.3. Leveraging Clojure's Strengths for Rich Error Information

Clojure, as WFL's implementation language, offers powerful features that can be harnessed to create a highly informative error reporting system. The community's consistent desire for improved error reporting in Clojure [9] highlights the importance of getting this right.

- **ex-data for Structured Payloads:** Clojure's ex-info function allows arbitrary data maps (referred to as ex-data) to be attached to thrown exceptions.[4] This is an ideal mechanism for including structured, WFL-specific contextual information directly within error objects. The new system will establish clear conventions and schemas for ex-data payloads associated with different categories of WFL errors, ensuring this rich information is also predictable and machine-parsable for advanced analytics and automated responses. While ex-data offers flexibility, its ad-hoc use can lead to inconsistent data structures. Therefore, defining these schemas or strong conventions is key to transforming ex-data from a mere feature into a cornerstone of a robust, analyzable error reporting system, aligning Clojure's flexibility with the systemic need for consistency.[1]
- **Custom Exception Types:** Clojure allows the definition of custom error types (e.g.,

using defrecord or deftype that implement clojure.lang.IExceptionInfo). This enables the creation of a WFL-specific exception hierarchy, facilitating more granular error handling and clearer categorization of issues.

- **Idiomatic Error Handling Patterns:** The system will encourage Clojure-idiomatic error handling. This includes effective use of try-catch-finally, careful consideration of error propagation to preserve context, and potentially exploring monadic error handling patterns (e.g., inspired by libraries like failjure [10] or concepts like Datomic's use of error maps and categories [5]) for specific scenarios where failures are common and expected outcomes rather than truly exceptional events. The goal is to make errors first-class, data-rich citizens within WFL.
- **Data-Oriented Approach:** Clojure's emphasis on data manipulation can be extended to error reporting, treating errors not just as signals but as rich data structures that can be queried, transformed, and analyzed.

By thoughtfully applying these Clojure-specific capabilities, the new error reporting system can provide a level of detail and utility that significantly surpasses generic error handling approaches.

# 4. Architectural Design: The Next-Generation WFL Error Reporting System

This section details the architectural blueprint for the new WFL error reporting system, covering the core components and strategies designed to meet the foundational principles outlined previously.

## 4.1. Unified Error Handling Framework for WFL

A unified error handling framework is the cornerstone of the new system. It ensures that errors are defined, captured, contextualized, and propagated consistently across all parts of WFL. This internal consistency is paramount, as the quality of data fed into downstream logging and monitoring systems directly dictates their effectiveness. Without standardizing how errors are managed within the application code, efforts to improve external observability will yield limited benefits due to inconsistent and poorly contextualized input.

### 4.1.1. Standardized Error Representation: Defining WFL Error Codes and Structured Payloads

To achieve consistency and machine-readability, a WFL-specific error catalog will be established. This catalog will feature:

- **Unique Error Identifiers (wflErrorId):** Each distinct type of error that WFL can recognize or generate will be assigned a stable, unique identifier (e.g., WFL-SRC-001, WFL-EXEC-CROMWELL-005). These IDs facilitate searching knowledge bases, linking to specific documentation, and enabling consistent programmatic handling or aggregation by monitoring systems.[1] The concept of human-readable error IDs, as explored in systems like jank [9], can also be considered for improved developer

ergonomics.

- **Human-Readable Message Templates:** Each wflErrorId will be associated with a clear, concise message template. These templates can include placeholders for dynamic contextual information, ensuring messages are informative yet standardized.
- **Structured Error Payloads (via ex-data):** A schema will be defined for the ex-data map associated with each wflErrorId or error category. This schema will detail expected contextual fields crucial for diagnosing WFL errors. Examples include:
    - workflowInstanceId: Identifier for the specific workflow run.
    - workflowDefinitionId: Identifier for the type of workflow.
    - executionStage: The WFL stage where the error occurred (e.g., "Source", "Executor-Submission", "Executor-Monitoring", "Sink").
    - relatedJobId: e.g., Cromwell job ID, GCS transfer ID.
    - inputParametersSnapshot: A sanitized snapshot of key input parameters relevant to the failure.
    - attemptNumber: If the operation was retried.
    - underlyingCause: Details of a wrapped, lower-level error. This approach is inspired by how systems like Datomic use error maps with namespaced keywords for extensible and generic error information.[5]
- **Severity Levels:** Each error type will have a default severity (e.g., DEBUG, INFO, WARNING, ERROR, CRITICAL) aligned with standard logging practices.[1]
- **Links to Documentation:** Where applicable, wflErrorId entries can include direct links to internal wiki pages or troubleshooting guides for that specific error.

### 4.1.2. Error Handling Across WFL Execution Phases: Strategies for Source, Executor, and Sink stages

Specific error handling logic will be defined for each primary WFL execution stage [3]:

- **Source Stage:**
    - **Error Types:** Failures in fetching input data/manifests (e.g., GCS access denied, file not found, malformed manifest structure), issues connecting to data repositories.
    - **Context to Capture:** Source URI, problematic file names, specific validation errors in manifests, permissions issues.
    - **Strategy:** Errors should be clearly distinguishable as input-related. For manifest validation, all errors should be collected and reported at once if possible, rather than failing on the first error.[1]
- **Executor Stage:** This stage involves submitting to and monitoring workflow execution engines like Cromwell.
    - **Error Types:** Cromwell submission failures (e.g., authentication, malformed WDL/inputs), Cromwell execution failures (job crashes, resource limits exceeded), timeouts waiting for Cromwell status, errors parsing Cromwell metadata.
    - **Context to Capture:** Cromwell workflow ID, Cromwell job ID(s), task name where failure occurred, relevant snippets from Cromwell logs/stderr, WFL's internal

representation of the workflow.
    - **Strategy:** A clear translation layer is needed to map Cromwell's diverse failure modes into standardized wflErrorIds. Implement robust polling and timeout logic for Cromwell interactions. Strategies for identifying transient vs. permanent Cromwell failures to inform retry logic.
- **Sink Stage:**
    - **Error Types:** Failures in delivering results (e.g., GCS write permissions, database connection errors for metadata updates), errors in triggering downstream notifications (e.g., JMS queue unavailable).
    - **Context to Capture:** Target output URIs, database connection details (excluding credentials), message queue identifiers.
    - **Strategy:** Ensure atomicity or idempotency where possible for sink operations to handle transient failures gracefully.

Across all stages, a **Correlation ID** generated at the initiation of a WFL workload (e.g., processing a set of samples) must be meticulously propagated. This ID will be included in all ex-data payloads and log messages, enabling the tracing of an error's path through WFL's internal stages and its interactions with external services.[1]

### 4.1.3. Managing Errors from External Services (Cromwell, GCS, DataRepo, JMS)

WFL's reliance on external services [3] necessitates resilient interaction patterns:

- **Error Translation:** External service errors (e.g., HTTP error codes from GCS, Cromwell API error responses, JMS exceptions) must be caught and translated into standardized WFL errors (with a specific wflErrorId and rich ex-data). The original external error details should be wrapped and included in the ex-data (e.g., under an :externalError key) for deep diagnostics.
- **Retry Mechanisms:** For transient errors from external services (e.g., temporary network issues, rate limiting), implement configurable retry logic with exponential backoff and jitter.[11] The Retry-After header, if provided by an API, should be respected.[1]
- **Circuit Breakers:** For services prone to prolonged outages, implement circuit breaker patterns [11] to prevent WFL from repeatedly overwhelming a failing service and to allow for faster failure detection for dependent operations.
- **Timeouts:** Configure appropriate timeouts for all calls to external services to prevent indefinite blocking.

### 4.1.4. Clojure-Specific Error Handling Patterns

To ensure WFL developers can effectively utilize the new framework, clear guidelines and idiomatic Clojure patterns will be promoted:

- **Effective try-catch Usage:**
    - Always catch specific exception types before generic ones (e.g., catch IOException e before catch Exception e). This principle, common in Java [1], applies equally to Clojure when dealing with Java interop or a hierarchy of custom Clojure exceptions.

    - Avoid empty catch blocks. If an exception is caught, it must be handled meaningfully: logged with full context, wrapped and re-thrown with additional context, or trigger a defined recovery action.
- **Maximizing ex-data:**
    - Mandate the use of ex-info and ex-data for all WFL-originated exceptions.
    - Provide utility functions or macros within WFL to simplify the creation of exceptions with standardized ex-data schemas (e.g., a function (wfl.error/throw-source-error error-id context-map cause)).
    - Document how to populate ex-data with dynamic contextual information available at the point of error (e.g., current workflow state, parameters being processed).
- **Custom WFL Exception Hierarchy (using defrecord or deftype):**
    - Define a hierarchy of Clojure error types that implement clojure.lang.IExceptionInfo (for ex-data support) and potentially clojure.lang.IDeref (if errors might carry lazily-computed details). Example:
      Clojure
      ```
      (defrecord WflSourceError [message data cause]
        clojure.lang.IExceptionInfo
        (getData [_] data)
        clojure.lang.IDeref
        (deref [_] {:message message :data data :cause cause}))
      ```
    - This allows for more precise catch clauses based on WFL-specific error categories.
- **Error Propagation and Wrapping:**
    - When catching an exception and re-throwing it as a WFL-specific error, the original exception (the cause) must be preserved and included in the ex-data of the new exception (or via Java's initCause if interoperability is key). This ensures the full causal chain is available for debugging.
- **Consideration of Monadic Error Handling (e.g., failjure):**
    - For certain WFL operations, particularly complex sequences or validation chains where failure is a common and expected outcome (not truly exceptional), a monadic error handling style using a library like failjure [10] or a custom Either-like construct could be beneficial. This can make code cleaner by avoiding deeply nested try-catch blocks.
    - However, this should be applied judiciously. Overuse for simple cases can add unnecessary complexity compared to standard exception handling. Clear guidelines will be provided on when this pattern is appropriate within WFL.

The following table provides examples of recommended Clojure error handling patterns for WFL:

**Table 2: Recommended Clojure Error Handling Patterns for WFL**

| Pattern | Description | Clojure Code Snippet Example (Illustrative) | Use Case in WFL | Benefits |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| **Rich ex-data Population** | Attaching a structured map of contextual information to exceptions using ex-info. | (throw (ex-info "Input manifest validation failed" {:wflErrorId "WFL-SRC-VALIDATE-001" :file/uri "gs://bucket/manifest.tsv" :details [...] :severity :ERROR})) | All WFL-originated errors, especially during input processing, workflow execution, or result sinking. | Provides detailed, machine-readable context directly with the error; facilitates targeted logging and analysis. |
| **Custom WFL Exception Type** | Defining specific error types (e.g., using defrecord) for different categories of WFL errors. | (defrecord WflExecutorError [msg ex-data] Throwable (init (proxy [Exception][msg] (getData ex-data)))) | Distinguishing between errors from Source, Executor, Sink stages, or specific external service interactions. | Allows for more granular catch clauses; improves code organization and error categorization. |
| **Error Wrapping with Cause** | Catching a lower-level exception and re-throwing it as a WFL-specific error, preserving the original cause. | (try... (catch SomeJavaException e (throw (ex-info "Cromwell submission failed" {:wflErrorId "WFL-EXEC-SUBMIT-002"...} e)))) | Handling errors from Java libraries or external services like Cromwell, GCS APIs. | Preserves the full stack trace and context of the original error; aids in deep-dive debugging. |
| **Conditional Monadic Usage** | Using an Either-like pattern or a library like failjure for sequences of operations where failure is a common, non-exceptional outcome. | (require '[failjure.core :as f]) (f/attempt-all [valid-input (validate-step1 input) processed (process-step2 valid-input)] (handle-success processed) (f/when-failed [err] (log-failure err))) | Complex data validation pipelines; multi-step operations where each step can return a "failure" result. | Can simplify code flow for non-exceptional failures; avoids try-catch for expected alternative paths. |
| **Standardized try-catch** | Following best practices for try-catch blocks, | (try... (catch MalformedURLException e | General error handling throughout WFL | Prevents accidental masking of |

| | | | | |
|---|---|---|---|---|
| | such as catching specific exceptions. | (handle-malformed-url e)) (catch IOException e (handle-io-failure e))) | code, especially around I/O or external calls. | unrelated errors; leads to more precise error recovery logic. |

## 4.2. Enhanced and Context-Aware Logging

Building upon WFL's existing JSON logging [3], the new system will significantly enhance log content and structure, especially for error events. This transforms logs from simple event records into rich diagnostic narratives, crucial for understanding complex failures in a distributed system like WFL. The goal is to drastically reduce the "how did I get here?" problem often faced by developers when encountering an error.[1]

### 4.2.1. Defining the Ideal WFL Error Log Event: Schema and Content

A detailed, standardized JSON schema for WFL error log entries will be defined. This schema will incorporate all essential components of an ideal error report [1] and be tailored to WFL's operational context. Each error log event will be a self-contained diagnostic package.

**Table 3: Anatomy of an Ideal WFL Error Log Event**

| Field Name | Data Type | Description/Purpose within WFL | Example Value (Illustrative) | Source of Data |
|---|---|---|---|---|
| timestamp | ISO8601 String | Precise UTC timestamp of when the error was logged (preferably when it occurred). | "2025-03-15T10:30:05.123Z" | Logging framework / System time |
| severity | String | Log level (e.g., "DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"). | "ERROR" | Error definition / Logger call |
| wflErrorId | String | Unique WFL-specific error code from the defined catalog (see 4.1.1). | "WFL-EXEC-CROMWELL-005" | ex-data from WFL exception |
| errorMessage | String | Concise, human-readable summary of the error, potentially generated from a | "Failed to retrieve status for Cromwell workflow abc-123 after 3 attempts." | ex-message / ex-data |

| | | | | |
|---|---|---|---|---|
| | | template associated with wflErrorId. | | |
| serviceContext.name | String | Name of the WFL service/component logging the error. | "wfl-api" or "wfl-executor-poller" | Application configuration / Environment |
| serviceContext.version | String | Version of the WFL application/service. | "1.2.3-beta" | Build information / Deployment manifest |
| host.name | String | Hostname or container ID where the error occurred. | "wfl-api-7f7d799d9b-xyz12" | Operating System / Kubernetes environment |
| host.ip | String | IP address of the host (if relevant and safe to log). | "10.0.1.25" | Operating System / Kubernetes environment |
| environment | String | Deployment environment (e.g., "development", "staging", "production"). | "production" | Application configuration / Environment variable |
| wfl.executionStage | String | The WFL execution phase where the error manifested (e.g., "Source", "Executor-Submission", "Sink"). | "Executor-Monitoring" | WFL internal context |
| wfl.workflowDefinitionId | String | Identifier for the type of workflow being processed (e.g., "ExternalExomeReprocessing"). | "WholeGenomeReprocessing_v2" | ex-data / WFL internal context |
| wfl.workflowInstanceId | String | Unique identifier for the specific WFL workload/workflow instance. | "wgs-batch-001-sample-05" | ex-data / WFL internal context |
| wfl.correlationId | String (UUID) | Unique ID tracing the entire WFL | "f47ac10b-58cc-4372-a567-0e02b2 | Generated at workload start, |

| | | | | |
|---|---|---|---|---|
| | | operation/workload across stages and services. | c3d479" | propagated via context / ex-data |
| externalSystem.name | String | Name of the external system involved in the error (e.g., "Cromwell", "GCS", "DataRepo"). | "Cromwell" | ex-data / WFL internal context |
| externalSystem.jobId | String | Identifier within the external system (e.g., Cromwell workflow ID). | "0a1b2c3d-4e5f-6a7b-8c9d-0e1f2a3b4c5d" | ex-data / WFL internal context |
| userContext.id | String | Identifier of the user who initiated the workload (if applicable and available). | "user@example.com" | ex-data / WFL authentication context |
| error.stackTrace.clojure | String | Full Clojure stack trace, obtained from the exception object (e.g., via clojure.repl/pst or similar). | "(... WflExecutorError... at wfl.executor$poll_cromwell.invokeStatic(executor.clj:123)...)" | Exception object |
| error.stackTrace.java | String | Full Java stack trace if the Clojure error was caused by a Java exception. | "(... java.net.SocketTimeoutException... at sun.nio.ch.SocketChannelImpl.read(SocketChannelImpl.java:380)...)" | Exception object (.getCause(), etc.) |
| error.causeChain | Array of Objects | Structured representation of the nested exception causes. Each object similar to the main error fields. | `` | Recursively processing ex-cause / .getCause() |

| | | | | |
|---|---|---|---|---|
| error.inputSnapshot | Object/String | Sanitized snapshot of key input parameters or data that led to the error. **Must be aggressively sanitized for PII/secrets.** | {"inputFile": "gs://bucket/data.bam", "reference": "hg38"} (sanitized) | ex-data / WFL internal context (carefully selected and sanitized) |
| error.customTags | Object | Key-value pairs for additional, flexible tagging and querying (e.g., {"project": "ProjectX", "priority": "high"}). | {"team": "alpha", "experimentId": "exp_987"} | ex-data / WFL internal context |
| error.links.documentation | String | URL to documentation for the wflErrorId. | "https://internal.wiki/wfl/errors/WFL-EXEC-CROMWELL-005" | Derived from wflErrorId / ex-data |
| error.links.trace | String | URL to a distributed trace in a tracing system, if available (using wfl.correlationId). | "https://tracing.example.com/traces/f47ac10b..." | Tracing system integration |

This detailed schema ensures that each error log is a rich source of diagnostic information, directly addressing the common frustration of missing context.[1]

### 4.2.2. Strategies for Capturing Rich Diagnostic Context

Populating the fields defined in the WFL Error Log Event schema requires careful instrumentation of WFL code:

- **Context Propagation:** WFL's correlationId and other relevant contextual data (like workflowInstanceId) must be passed through call chains, potentially using Clojure's dynamic vars (binding) for some thread-local context, or explicitly passed as arguments to functions involved in a workload's processing. WFL's util.clj [3] could house utilities for managing and accessing this propagated context.
- **Data Sanitization:** A critical aspect is the sanitization of sensitive information before logging.[1] Input parameters (error.inputSnapshot), user details, and any other potentially sensitive data must be filtered or redacted. This should be a non-negotiable step, with clear rules and utility functions for sanitization.
- **Stack Trace Generation:** Ensure that full Clojure stack traces are captured. If errors are

wrapped, the entire cause chain must be preserved and logged, possibly using utilities that can serialize the nested structure of ex-data from each exception in the chain.
- **Automated Context Enrichment:** The logging framework or dedicated WFL logging utilities should automatically enrich log events with available ambient context like serviceContext.name, serviceContext.version, host.name, and environment.

#### 4.2.3. Dynamic Log Level Management and Configuration

WFL's existing dynamic log level API [3] is a good foundation. Enhancements could include:
- **Granular Control:** Allow log levels to be set not just globally, but also on a per-module (Clojure namespace) basis, or even temporarily for a specific workflowDefinitionId or workflowInstanceId. This would enable targeted, verbose debugging for specific issues without flooding the logs for the entire system.
- **Environment-Specific Defaults:** Define clear default logging configurations for different environments (e.g., DEBUG or INFO for development and testing, WARNING or ERROR for production).
- **Audit Logging for Level Changes:** Log any changes made to logging levels via the API for auditing purposes.

#### 4.2.4. Integration with Centralized Logging (Optimizing Stackdriver Usage)

The new, richly structured error logs will continue to be sent to Google Stackdriver (Cloud Logging). Optimization strategies include:
- **Proper Field Indexing:** Ensure that key fields from the new schema (e.g., wflErrorId, wfl.workflowInstanceId, wfl.correlationId, wfl.executionStage) are correctly parsed and indexed by Stackdriver to enable fast and efficient querying and filtering.
- **Leveraging Stackdriver Features:** Utilize Stackdriver's capabilities for log-based metrics, custom dashboards, and advanced log exploration based on the richer data.
- **Evaluation Against Dedicated Tools:** While Stackdriver is the incumbent, periodically evaluate if its capabilities for error analysis, grouping, and presentation are sufficient, or if a dedicated error tracking tool (discussed in 4.3) would offer significant advantages for WFL's specific needs. The decision should be based on a cost-benefit analysis and the WFL team's operational preferences.

## 4.3. Intelligent Error Monitoring, Alerting, and Observability

The new system aims to transition WFL from primarily reactive log-based alerting [3] to a more intelligent and proactive error monitoring posture. This shift is crucial for minimizing downtime, reducing developer toil spent on "firefighting," and ultimately supporting WFL's goal of optimized, reliable workflow execution.[2] This involves implementing or integrating tools and strategies for real-time detection, intelligent processing of errors, actionable alerting, and comprehensive observability.

#### 4.3.1. Real-time Error Detection, Aggregation, and Intelligent Grouping

Moving beyond simple severity>=ERROR alerts requires more sophisticated processing:

- **Real-time Capture and Processing:** Error log events (as defined in 4.2.1) should be streamed to a monitoring system or processed in near real-time.
- **Intelligent Error Grouping:** This is a cornerstone feature for reducing noise and improving signal quality.[1] The system should automatically group occurrences of the same underlying error into a single "issue" or "incident." Grouping logic can be based on:
    - wflErrorId.
    - Stack trace similarity (fingerprinting the call stack).
    - A combination of error message patterns and key contextual fields. This dramatically reduces alert storms for recurring problems and clearly highlights the frequency and impact of unique bugs, directly combating "alert fatigue".[1]
- **De-duplication:** Ensure that multiple instances of the exact same error event (e.g., from retries or rapid occurrences) do not trigger redundant notifications if they are part of an already identified and grouped issue.
- **First Seen/Last Seen Tracking:** For each grouped issue, track when it was first detected and when the most recent occurrence happened. This helps in understanding the lifecycle of an error.

### 4.3.2. Proactive and Actionable Alerting Strategies

Alerts must be timely, relevant, and provide clear pathways to action:

- **Alerting Criteria:**
    - **New Error Detection:** Alert when a previously unseen wflErrorId (or a new unique error group) appears in a specific environment (especially production).
    - **Error Rate Spikes:** Alert if the frequency of a specific wflErrorId, errors from a critical WFL component (e.g., Cromwell poller), or errors affecting a particular workflowDefinitionId exceeds a dynamically determined baseline or a predefined threshold. Anomaly detection algorithms can be employed here.
    - **Impact-Based Alerting:** Trigger alerts if an error (or group of errors) affects a significant number of workflow instances, specific high-priority workflows, or a notable percentage of the total active workload.
    - **Regressions:** Alert if an error that was previously marked as "resolved" reappears, especially after a new deployment.
- **Alert Content and Routing:**
    - Alerts must contain a concise summary of the issue, the wflErrorId, severity, affected component/workflow, and a direct link to the grouped error details in the monitoring system.
    - Links to relevant logs in Stackdriver (filtered by wfl.correlationId or wflErrorId) and pertinent observability dashboards should also be included.
    - Implement severity-based routing: Critical errors (e.g., system-wide outage, major data corruption risk) could be routed to on-call systems like PagerDuty, whereas less severe but important errors or warnings might go to a dedicated Slack channel or email distribution list.[1]

- **Configurability:** Alerting rules and notification channels must be easily configurable by the WFL operations team.

### 4.3.3. Designing Observability Dashboards for WFL Errors

Visual dashboards are essential for providing an at-a-glance overview of WFL's error landscape and identifying trends or systemic issues.[1] Key dashboards should include:

- **Overall Error Health Dashboard:**
  - Total error rate over time (e.g., errors per hour/day).
  - Error rate broken down by severity.
  - Count of new unique errors detected recently.
  - Top N most frequent wflErrorIds.
  - MTTA (Mean Time To Acknowledge) and MTTR (Mean Time To Resolution) for critical errors.
- **WFL Component/Stage Performance Dashboard:**
  - Error rates per WFL component (e.g., wfl-api, wfl-source-manager, wfl-executor-cromwell) or execution stage (Source, Executor, Sink).
  - Latency and error rates for interactions with key external services (Cromwell, GCS, DataRepo).
- **Workflow-Specific Error Dashboard:**
  - Ability to filter error metrics by workflowDefinitionId or workflowInstanceId.
  - Error breakdown by step or task within common workflow types.
- **Release Quality Dashboard:**
  - Correlation of error rates and new error introductions with WFL deployments/releases.
  - Comparison of error profiles between different application versions.

These dashboards should leverage the rich contextual data from the standardized error log events, allowing for drill-down capabilities from aggregated views to specific error instances or related log entries.

By implementing these intelligent monitoring, alerting, and observability features, WFL can move towards a more proactive operational model. This not only helps in fixing errors faster but also in identifying and mitigating systemic weaknesses before they lead to widespread impact, a crucial step in achieving the high levels of reliability and efficiency envisioned for WFL.[2]

## 4.4. API Error Reporting Standards (for WFL's external interfaces)

If WFL exposes external APIs—for instance, to submit workloads, query status, or manage configurations, as suggested by the presence of api/handlers.clj and api/routes.clj [3]—it is imperative that these APIs communicate errors in a standardized, developer-friendly, and machine-readable manner. Poor API error handling is a significant source of frustration for consuming developers and can lead to brittle, unreliable integrations.[1]

The new system will adopt established industry standards for HTTP API error responses:

- **Adherence to RFC 9457 or AIP-193:**

    - **RFC 9457 ("Problem Details for HTTP APIs"):** This standard provides a way to carry machine-readable details of errors in an HTTP response, using a JSON object with predefined members like type, title, status, detail, and instance. This is the preferred standard for broad interoperability.
    - **Google's AIP-193 (Error Model):** Offers similar guidelines for structured error responses, often used within the Google Cloud ecosystem. Given WFL's GCP deployment, this is also a strong candidate.
    - The chosen standard will be applied consistently across all WFL API endpoints.
- **Accurate HTTP Status Codes:**
    - **4xx Client Errors:** Used for errors attributable to the client's request (e.g., 400 Bad Request for malformed syntax or invalid parameters, 401 Unauthorized for missing/invalid credentials, 403 Forbidden for insufficient permissions, 404 Not Found for unknown resources, 422 Unprocessable Entity for semantic errors in a syntactically valid request, 429 Too Many Requests for rate limiting).[1]
    - **5xx Server Errors:** Used for errors originating on the server-side (e.g., 500 Internal Server Error for unexpected conditions, 502 Bad Gateway if WFL relies on an upstream service that failed, 503 Service Unavailable for temporary unavailability).[1]
- **Consistent JSON Error Body:** Regardless of the specific standard chosen, the error response body will be JSON and include:
    - type: A URI that identifies the problem type. This URI should resolve to human-readable documentation explaining the error code (e.g., linking to a page describing the wflErrorId).
    - title: A short, human-readable summary of the problem type.
    - status: The HTTP status code (repeated in the body for convenience).
    - detail: A human-readable explanation specific to this occurrence of the problem.
    - instance: A URI that identifies the specific occurrence of the problem. This might be a unique error instance ID.
    - wflErrorId: The WFL-specific error code (from 4.1.1) to allow clients to programmatically identify the error.
    - requestId (or correlationId): The unique ID for the request, allowing clients to correlate this error with their logs and WFL's logs.
    - errors (optional, for validation): If a request results in multiple validation errors (e.g., on a complex input object), this field can be an array of objects, each detailing a specific field error (field name, message, invalid value). This practice of returning all validation errors at once is highly recommended.[1]
- **Security of API Error Responses:** API error responses must not leak sensitive internal details such as internal stack traces, raw database error messages, or system configuration paths. Information should be curated to be helpful to the API consumer without compromising WFL's security.

Standardizing WFL's API error responses not only enhances the experience for human developers integrating with WFL but also significantly improves the robustness and resilience

of automated clients (e.g., other microservices, CI/CD pipelines, or scripts). Automated systems depend heavily on predictable, machine-parsable error responses to implement their own sophisticated error handling logic, such as conditional retries, fallbacks, or custom alerting. Inconsistent or uninformative API errors from WFL would compel client developers to implement complex and fragile parsing logic, increasing integration efforts and the likelihood of misinterpreting error conditions. By adopting a clear standard like RFC 9457, WFL promotes better ecosystem stability and more reliable automated interactions.

## 4.5. Security by Design in Error Communication

Security considerations must be an integral part of the design of all error handling and reporting mechanisms.[1] An insecure error reporting system can inadvertently transform a diagnostic tool into an attack vector, potentially exposing sensitive information or system vulnerabilities. Given that WFL processes potentially sensitive scientific data (e.g., genomic data [3]) and operates within a cloud environment, a "security by design" approach to error communication is not merely a best practice but a critical risk mitigation strategy. Failure in this area could have significant compliance, data integrity, and reputational implications. The following security principles will be embedded in the WFL error reporting system:

- **Prevent Sensitive Data Leakage:**
    - **Rigorous Sanitization:** All data, especially user-supplied inputs, configuration details, personal identifiable information (PII), or any data that might be part of scientific payloads, must be aggressively filtered, redacted, or masked before being included in error messages, log entries (especially the error.inputSnapshot field), or API responses.[1]
    - **Stack Trace Review:** Full stack traces, while invaluable for internal debugging, should not be exposed to external clients in production API responses. If parts of stack traces are logged, they must be reviewed to ensure they do not contain sensitive information like file paths revealing internal directory structures or data embedded in variable names.
    - **Controlled Verbosity:** External-facing error messages must be concise and avoid revealing internal system architecture, library versions, or database schemas. Internal logs can afford to be more verbose for debugging by trusted developers, but the information exposed externally must be carefully curated.[1]
- **Sanitize Inputs and Outputs for Error Generation:**
    - **Input Validation:** Rigorous input validation at API boundaries and internal component interfaces can prevent malformed or malicious inputs from triggering errors that might, in turn, expose vulnerabilities or reveal excessive system information.
    - **Output Encoding/Sanitization:** If error messages or details are ever rendered in an HTML context (e.g., an internal WFL admin UI), any user-supplied data incorporated into these messages must be properly encoded or sanitized to prevent cross-site scripting (XSS) vulnerabilities.
- **Generic Authentication and Authorization Errors:**

    - For failures related to authentication (e.g., invalid credentials) or authorization (e.g., insufficient permissions), WFL APIs and internal systems should return generic error messages (e.g., "Invalid credentials," "Access denied," or a standard HTTP 401 Unauthorized / 403 Forbidden).
    - Avoid specific messages like "User X not found" or "Invalid password for user Y," as these can confirm the existence (or non-existence) of user accounts, aiding attackers in account enumeration or targeted attacks.[1]
- **Control Information Disclosure from Error Probing:**
    - Attackers often probe systems by intentionally triggering errors to gather intelligence about the system's architecture, underlying technologies (server types, framework versions), database structures, or internal logic.[1]
    - The error reporting system must be designed to minimize the information an attacker can glean from such probes. This reinforces the need for generic, non-revealing error messages in external responses.
- **Secure Configuration of Logging and Monitoring Tools:**
    - Access to centralized logging systems (Stackdriver) and monitoring dashboards containing error information must be appropriately restricted based on roles and responsibilities.
    - Ensure that any credentials used by WFL to send data to these systems have the minimum necessary permissions.

A multi-layered approach to security in error reporting is essential, encompassing secure default configurations in WFL's frameworks and libraries, robust tools and processes for data sanitization and filtering within the error reporting pathways, and continuous developer awareness and adherence to these security best practices. Regular security reviews and penetration testing of WFL should include an assessment of how the application handles and reports errors, particularly focusing on potential information leakage.

# 5. Implementation Strategy and Recommendations

The successful deployment of the new error reporting system for WFL requires a well-planned implementation strategy that considers phased rollout, appropriate tooling, developer enablement, and mechanisms for continuous improvement. The goal is to deliver a technically sound system that is also effectively adopted and utilized by the WFL team. The success of this initiative hinges as much on its thoughtful implementation and adoption strategy as on its architectural design; a technically brilliant system that is difficult for developers to use or poorly integrated will fail to achieve its objectives.[1]

## 5.1. Phased Rollout Approach

A gradual, phased rollout is recommended to manage risk, gather feedback, and allow for iterative refinement:

- **Phase 1: Foundation and Pilot (Target: 2-3 Months)**
    - **Focus:** Implement the Unified Error Handling Framework (Section 4.1) within a single, critical WFL module or for a specific, well-understood workload type (e.g.,

a common reprocessing pipeline).
    - **Activities:**
        - Define the initial WFL error catalog (wflErrorIds, message templates, ex-data schemas) for the pilot module/workload.
        - Develop core Clojure utilities for creating and throwing standardized WFL exceptions with rich ex-data.
        - Implement the WFL Error Log Event schema (Section 4.2.1) for the pilot scope.
        - Ensure basic integration with Stackdriver for these new structured error logs.
        - Establish initial developer guidelines for the pilot.
    - **Goal:** Validate the core error handling framework and logging schema in a limited context. Gather early developer feedback.
- **Phase 2: Enrichment and Broader Adoption (Target: 3-4 Months)**
    - **Focus:** Expand the rollout of the error handling framework and enhanced logging to more WFL modules and workload types.
    - **Activities:**
        - Refine and expand the WFL error catalog based on learnings from Phase 1.
        - Implement strategies for capturing richer diagnostic context across more components (Section 4.2.2), including robust correlation ID propagation.
        - Begin developing initial observability dashboards in Stackdriver (or chosen tool) based on the new error log data (Section 4.3.3).
        - Conduct broader developer training on the new error handling and logging practices.
    - **Goal:** Achieve wider coverage of the new system within WFL. Improve the depth of contextual information captured. Provide initial visibility through dashboards.
- **Phase 3: Intelligence, API Standardization, and Optimization (Target: 3-4 Months)**
    - **Focus:** Implement intelligent error monitoring features and standardize API error reporting.
    - **Activities:**
        - Integrate or configure intelligent error grouping, de-duplication, and proactive alerting strategies (Sections 4.3.1, 4.3.2). This may involve deeper configuration of Stackdriver or integration of a dedicated error monitoring tool.
        - If WFL exposes external APIs, implement standardized error responses according to RFC 9457 or AIP-193 (Section 4.4).
        - Implement comprehensive security measures for error communication (Section 4.5).
        - Optimize logging performance and Stackdriver query efficiency.
    - **Goal:** Transition to proactive error management. Ensure external API consumers benefit from standardized error reporting. Harden security aspects.
- **Continuous Iteration:** Each phase should conclude with a review and feedback session. The error reporting system is not static; it will require ongoing refinement based

on WFL's evolution, new error patterns discovered, and developer input.[7]

## 5.2. Recommended Tooling and Libraries

The choice of tooling should balance capabilities, cost, developer familiarity, and operational overhead.

- **Core Error Handling (Clojure):**
  - **Standard Library:** Leverage Clojure's built-in try-catch-finally, ex-info, and ex-data as the primary mechanisms.
  - **Custom WFL Utilities:** Develop a small, focused WFL-internal Clojure library (e.g., wfl.error-utils) to:
    - Provide helper functions/macros for creating standardized WFL exceptions with pre-defined ex-data structures.
    - Offer utilities for sanitizing data before inclusion in ex-data or logs.
    - Assist in managing and propagating correlation IDs and other contextual data.
  - **Monadic Libraries (e.g., failjure [10]):** Evaluate failjure or similar libraries for specific, complex error-prone flows where they can genuinely simplify code (e.g., long validation chains). Caution against widespread, premature adoption if it overcomplicates simpler error handling scenarios. The default should be standard Clojure exceptions unless a clear benefit is demonstrated for a monadic approach.
- **Logging:**
  - **Facade:** Continue using clojure.tools.logging as the logging facade.
  - **Backend:** Utilize an appropriate backend that integrates seamlessly with Google Stackdriver (Cloud Logging), ensuring structured JSON output. The focus will be on enhancing the *content* and *structure* of the log messages as per Section 4.2.1, rather than changing the fundamental logging libraries unless a compelling reason arises.
- **Monitoring, Alerting, and Error Aggregation:**
  - **Option A: Enhance Google Stackdriver (Cloud Monitoring/Logging):**
    - **Pros:** Leverages existing GCP infrastructure [3]; potentially lower initial cost if already part of GCP commitment; unified platform for logs and metrics.
    - **Cons:** May require significant custom configuration to achieve intelligent grouping and sophisticated alerting comparable to dedicated tools; UI/UX for error analysis might be less specialized than dedicated tools.
    - **Implementation:** Create advanced log-based metrics from the new rich error logs. Develop sophisticated alert policies in Cloud Monitoring. Build custom dashboards.
  - **Option B: Dedicated Error Monitoring Tool (e.g., Sentry, Bugsnag, Rollbar):**
    - **Pros:** Offer advanced features out-of-the-box, such as intelligent error grouping, de-duplication, release tracking, sophisticated UIs for error analysis, and often better SDKs for capturing rich client-side context.[1] Can

significantly accelerate the implementation of intelligent monitoring.
        - **Cons:** Additional cost (licensing fees); introduces another tool to the WFL operational stack; requires integration effort.
        - **Implementation:** Integrate the chosen tool's SDK into WFL. Configure it to ingest WFL error reports.
    - **Recommendation:** Start by maximizing Stackdriver's capabilities (Option A), especially during Phases 1 and 2, due to its existing integration. Concurrently, conduct a thorough evaluation (Proof of Concept) of one or two leading dedicated error monitoring tools. If the benefits (e.g., significantly better grouping, developer workflow integration, reduced configuration effort) outweigh the costs and operational overhead, consider adopting a dedicated tool in Phase 3 or as a subsequent enhancement. The decision should be driven by WFL's specific needs for error analysis depth and the team's operational capacity.

## 5.3. Developer Onboarding: Guidelines, Best Practices, and Training

Effective developer onboarding is crucial for the successful adoption and consistent utilization of the new error reporting system. If instrumenting code for high-quality error reporting is perceived as difficult or an afterthought, developers are less likely to invest the necessary effort, thereby undermining the system's value.[1]

- **Comprehensive Documentation:**
    - **WFL Error Reporting Guide:** A dedicated section in WFL's developer documentation covering:
        - The WFL error catalog (list of wflErrorIds, their meanings, severities).
        - Detailed schemas for ex-data payloads for common error categories.
        - Conventions for logging errors and the structure of WFL error log events.
        - How to use any new WFL error utility libraries or macros.
        - Best practices for error propagation and context preservation.
        - Guidelines on data sanitization.
- **Best Practices and "Dos and Don'ts":**
    - Provide a clear, concise list of best practices, for example:
        - DO use specific wflErrorIds when throwing exceptions.
        - DO populate ex-data with all relevant contextual fields as per the defined schemas.
        - DO preserve the original cause when wrapping exceptions.
        - DON'T catch generic Exception without specific handling or re-throwing with WFL context.
        - DON'T log sensitive information without proper sanitization.
        - DO ensure correlation IDs are propagated and included in error context.
- **Training and Workshops:**
    - Conduct interactive training sessions or workshops for all WFL developers to introduce the new system, its rationale, and practical usage.
    - Include hands-on exercises for refactoring existing error handling or instrumenting new code.

- **Code Examples and Templates:**
    - Provide readily available code snippets and templates demonstrating how to correctly throw, catch, log, and handle WFL errors according to the new standards.
    - Include examples in a WFL reference application or a dedicated examples repository.
- **Code Reviews:** Incorporate adherence to the new error reporting standards as a checklist item in code reviews.

## 5.4. Establishing Continuous Feedback Loops for System Refinement

The error reporting system is not a "set it and forget it" solution. It must evolve with WFL and the changing understanding of its failure modes. Establishing feedback loops is essential for its ongoing effectiveness and relevance.[1]

- **Developer Feedback Channels:**
    - Create clear channels (e.g., a dedicated Slack channel, a specific Jira component) for developers to provide feedback on the error reporting system itself. This includes reporting:
        - Unhelpful or missing wflErrorIds.
        - Insufficient context in error reports or logs.
        - Noisy or un actionable alerts.
        - Difficulties in using the error handling utilities.
        - Suggestions for improving error grouping or dashboard visualizations.
- **Monitoring System Efficacy:**
    - Track key metrics related to the error reporting system's performance, such as:
        - Mean Time To Acknowledge (MTTA) for new error alerts.
        - Mean Time To Resolution (MTTR) for errors reported by the system.
        - Error recurrence rates (how often "resolved" errors reappear).
        - Developer satisfaction surveys regarding the error reporting system.
    - If these metrics are not improving or are degrading, it indicates that the error reporting system may require further refinement.
- **Regular Review and Refinement Meetings:**
    - Schedule periodic (e.g., quarterly) review meetings involving key WFL developers, operations staff, and architects.
    - Agenda items should include:
        - Reviewing the top occurring errors and their impact.
        - Analyzing trends in error rates and types.
        - Discussing developer feedback received.
        - Identifying areas for improvement in error messages, ex-data schemas, linked documentation, logging verbosity, alert configurations, or dashboard utility.
        - Planning updates to the error catalog or utility libraries.

By fostering a culture of continuous improvement around error reporting, WFL can ensure that this system remains a powerful asset that adapts to the evolving needs of the platform and its

development team.

# 6. Conclusion

The design outlined in this report proposes a significant evolution for Workflow Launcher's error reporting capabilities, moving from an implicit, reactive state to a proactive, developer-centric, and highly observable system. The strategic importance of such a system for WFL cannot be overstated. As WFL tackles increasingly complex scientific workloads and aims for "smart factory" levels of operational efficiency and process safety [2], the ability to rapidly detect, diagnose, and remediate errors becomes a critical enabler of success.
The proposed architecture addresses the identified weaknesses in the current system by introducing:

- A **Unified Error Handling Framework** that standardizes error definitions, payloads, and propagation within WFL, leveraging Clojure's ex-data for rich, structured information.
- **Enhanced and Context-Aware Logging** that transforms log entries into comprehensive diagnostic dossiers, complete with correlation IDs and detailed WFL-specific context.
- **Intelligent Error Monitoring and Alerting** designed to reduce noise, provide actionable insights through smart grouping and proactive notifications, and offer deep observability via targeted dashboards.
- **Standardized API Error Reporting** to improve the experience for external consumers and the robustness of integrations.
- A foundational commitment to **Security by Design**, ensuring that error communications do not compromise sensitive data or system integrity.

This design is deeply rooted in WFL fundamentals—its Clojure foundation, staged architecture, and cloud-native deployment—while drawing extensively from industry best practices for developer-centric error reporting.[1] The anticipated benefits are substantial:

- **Significantly Improved Developer Productivity:** Clear, contextual, and actionable error reports will drastically reduce the time developers spend on debugging and root cause analysis.
- **Faster Issue Resolution:** Enhanced visibility and intelligent alerting will lead to quicker acknowledgment and resolution of errors (reduced MTTA/MTTR).
- **Enhanced System Reliability and Stability:** Proactive identification of error trends and systemic issues will enable preventative measures, leading to a more robust and stable WFL platform.
- **Better Operational Insight:** Comprehensive dashboards and queryable error data will provide valuable insights into WFL's operational health and failure modes.
- **Stronger Foundation for WFL's Strategic Goals:** A mature error reporting system is a prerequisite for achieving high levels of automation, process optimization, and safety in complex workflow environments.

The implementation of this new error reporting system represents a strategic investment in WFL's future. It is an investment in the development team's efficiency and satisfaction, in the quality and reliability of the software, and in WFL's capacity to deliver on its ambitious

operational objectives. It is therefore strongly recommended that WFL stakeholders commit the necessary resources to implement this vital system, following the proposed phased approach and fostering a culture of continuous improvement around error management.

**Works cited**


1. error report.pdf
2. WFL Operational Data Acquisition System - Gear Technology, accessed May 30, 2025, https://www.geartechnology.com/videos/wfl-operational-data-acquisition-system
3. Workflow Launcher - Broad Institute, accessed May 30, 2025, https://broadinstitute.github.io/wfl/
4. Error handling in clojure - Beginners - ClojureVerse, accessed May 30, 2025, https://clojureverse.org/t/error-handling-in-clojure/1877
5. Error Handling - Datomic Documentation, accessed May 30, 2025, https://docs.datomic.com/api/error-handling.html
6. Software - WFL Millturn Technologies, accessed May 30, 2025, https://www.wfl.at/en/software
7. Execution Phase - Adaptive Acquisition Framework, accessed May 30, 2025, https://aaf.dau.edu/aaf/software/execution-phase/
8. Fundamentals of Software Architecture: An Engineering Approach - Amazon.com, accessed May 30, 2025, https://www.amazon.com/Fundamentals-Software-Architecture-Comprehensive-Characteristics/dp/1492043451
9. Can jank beat Clojure's error reporting?, accessed May 30, 2025, https://jank-lang.org/blog/2025-03-28-error-reporting/
10. adambard/failjure: Monadic error utilities for general use in Clojure(script) projects - GitHub, accessed May 30, 2025, https://github.com/adambard/failjure
11. Error Handling | Clojure Patterns, accessed May 30, 2025, https://clojurepatterns.com/6/