Collatz Engine: A Public Computational System for Sequential Evaluation of the Collatz Map

Technical Report 001

Amaete Umanah

June 2026

Abstract

Collatz Engine is a persistent computational system for sequentially evaluating the Collatz map over positive integers and recording finite computational behavior, including trajectories, stopping times, peak values, and record events. This report documents the system architecture, computation method, persistence model, data structure, and public reporting layer. The report does not present a proof of the Collatz conjecture. It describes a finite computational observatory and the conditions under which its results should be interpreted.

Contents

1 Introduction **6**

1.1 The Collatz Map 6

1.2 The $3n + 1$ Problem 6

1.3 Computation and Its Limits 7

1.4 Purpose and Scope of This Report 7

2 System Overview **10**

2.1 Definition of Collatz Engine 10

2.2 Sequential Evaluation Model 10

2.3 Core System Components 11

2.4 Public Observability Requirements 12

2.5 State Persistence as a Design Requirement 13

2.6 System Boundaries and Non-Goals 13

2.7 Interpretation of the System 13

3 Mathematical Definitions and Computed Quantities **15**

3.1 Purpose of the Definitions 15

3.2 Iteration Function 15

3.3 Trajectory . . . 16
3.4 Total Stopping Time . . . 16
3.5 Trajectory Length . . . 17
3.6 Peak Value . . . 17
3.7 Known Cycle Containing 1 . . . 18
3.8 Record Events . . . 18
3.9 Per-Integer Result . . . 18
3.10 Aggregate Engine State . . . 19
3.11 Finite Verification Statement . . . 19

4 Computation Method **21**
4.1 Purpose of the Computation Method . . . 21
4.2 Starting Integer Selection . . . 21
4.3 Trajectory Evaluation . . . 22
4.4 Operational Pseudocode . . . 22
4.5 Per-Integer Result Generation . . . 23
4.6 Aggregate Record Updates . . . 24
4.7 Batch Execution . . . 24
4.8 Handling Intermediate Values . . . 25
4.9 Termination and Failure Modes . . . 25
4.10 Determinism and Recomputability . . . 26
4.11 Summary . . . 26

5 Persistent Engine State **28**
5.1 Purpose of Persistent State . . . 28
5.2 Database as Source of Truth . . . 28
5.3 Engine State Variables . . . 29
5.4 State Transition Model . . . 29
5.5 Resume Behavior After Interruption . . . 30
5.6 Timestamp-Based Runtime Accounting . . . 30
5.7 Heartbeat and Staleness Detection . . . 31
5.8 Concurrency and Locking . . . 32
5.9 Atomicity of Progress Updates . . . 32
5.10 Public State Versus Internal State . . . 32
5.11 Failure Recovery . . . 33
5.12 Summary . . . 34

6 Data Model and Storage Design **35**
6.1 Purpose of the Data Model . . . 35
6.2 Primary Tables . . . 35

6.3 Engine State Table . . . 35
6.4 Results Table . . . 37
6.5 Record Events Table . . . 38
6.6 Activity Logs Table . . . 38
6.7 Run Intervals and Runtime Accounting . . . 39
6.8 Relationships Between Tables . . . 40
6.9 Constraints and Indexes . . . 40
6.10 Storage of Large Integers . . . 41
6.11 Retention and Aggregation . . . 41
6.12 Data Access and Public Reporting . . . 42
6.13 Summary . . . 42

7 Integrity, Reproducibility, and Auditability 43
7.1 Purpose of Integrity Controls . . . 43
7.2 Deterministic Computation . . . 43
7.3 Definition Consistency . . . 44
7.4 Sequential Coverage Assumption . . . 44
7.5 Duplicate Prevention . . . 45
7.6 Gap Detection . . . 45
7.7 Crash Recovery and Durable Progress . . . 46
7.8 Record Event Integrity . . . 46
7.9 Dashboard Consistency . . . 47
7.10 Independent Recalculation . . . 47
7.11 Audit Logs . . . 48
7.12 Reconciliation Procedures . . . 49
7.13 Integrity Boundaries . . . 49
7.14 Summary . . . 49

8 Public Dashboard and Visualization Layer 51
8.1 Purpose of the Public Dashboard . . . 51
8.2 Dashboard as a Reporting Surface . . . 51
8.3 Core Dashboard Fields . . . 52
8.4 Persistent Runtime Clock . . . 52
8.5 Current Number and Checked Range . . . 53
8.6 Total Checked . . . 53
8.7 Longest Trajectory Display . . . 54
8.8 Highest Peak Display . . . 54
8.9 Throughput Reporting . . . 54
8.10 Trajectory Visualization . . . 55
8.11 Heatmaps and Aggregate Visualizations . . . 55

8.12 Record Event Feed . . . 56
8.13 Status, Staleness, and Error Display . . . 56
8.14 Figure Placeholder for Dashboard . . . 57
8.15 Interpretation Constraints . . . 57
8.16 Summary . . . 58

9 Current Computational Status **59**
9.1 Purpose of This Status Section . . . 59
9.2 Dashboard Snapshot . . . 59
9.3 Reported Public Dashboard Values . . . 60
9.4 Checked Range Interpretation . . . 60
9.5 Trajectory and Visualization Status . . . 61
9.6 Result Tables and Record Sections . . . 61
9.7 Operational Status . . . 62
9.8 Interpretation of Current Results . . . 62
9.9 Data Quality Notes . . . 62
9.10 Summary . . . 63

10 Limitations **64**
10.1 Purpose of This Section . . . 64
10.2 Finite Computation Is Not Proof . . . 64
10.3 Implementation Correctness . . . 64
10.4 Numeric Precision and Overflow . . . 65
10.5 Persistence and State Consistency . . . 65
10.6 Throughput Constraints . . . 66
10.7 Single-Worker Architecture . . . 66
10.8 Dashboard Interpretation . . . 66
10.9 Visualization Limits . . . 67
10.10 Data Retention Limits . . . 67
10.11 External Verification Limits . . . 67
10.12 Operational Downtime . . . 67
10.13 Scope of This Report . . . 68
10.14 Summary . . . 68

11 Future Work **69**
11.1 Purpose of Future Work . . . 69
11.2 Distributed Workers . . . 69
11.3 Range Assignment and Verification . . . 70
11.4 Independent Verifier Tooling . . . 70
11.5 Public Data Exports . . . 71

11.6 Public API Access . . . 72
11.7 Improved Runtime Accounting . . . 72
11.8 Record Event Analysis . . . 73
11.9 Visualization Extensions . . . 73
11.10 Code and Schema Publication . . . 74
11.11 Archival Releases and Citation . . . 74
11.12 Future Reports . . . 75
11.13 Summary . . . 75

12 Conclusion **76**
12.1 Summary of the System . . . 76
12.2 Computation and Interpretation . . . 76
12.3 Persistence and Auditability . . . 76
12.4 Role of the Dashboard . . . 77
12.5 Limits of the Current System . . . 77
12.6 Path Forward . . . 78
12.7 Closing Statement . . . 78

1 Introduction

1.1 The Collatz Map

The Collatz map is a function on the positive integers. In its standard form, it sends an integer n to $n/2$ when n is even, and to $3n+1$ when n is odd:

$$T(n) = \begin{cases} n/2, & \text{if } n \equiv 0 \pmod 2, \\ 3n+1, & \text{if } n \equiv 1 \pmod 2. \end{cases}$$

Starting from a positive integer n, repeated application of T produces a sequence

$$n,\ T(n),\ T^2(n),\ T^3(n),\ \ldots$$

This sequence is usually called the trajectory, orbit, or stopping sequence of n, depending on context. This report uses the term *trajectory*.

For example, the trajectory beginning at $n = 7$ is

$$7,\ 22,\ 11,\ 34,\ 17,\ 52,\ 26,\ 13,\ 40,\ 20,\ 10,\ 5,\ 16,\ 8,\ 4,\ 2,\ 1.$$

After reaching 1, the sequence enters the repeated cycle

$$1,\ 4,\ 2,\ 1.$$

The central question is whether this behavior occurs for every positive integer. The Collatz conjecture asserts that, for every positive integer n, the trajectory generated by repeated application of T eventually reaches 1. No general proof is known, and no counterexample is known [1].

1.2 The $3n+1$ Problem

The Collatz conjecture is also commonly called the $3n+1$ problem. It is simple to state but has resisted general proof. The difficulty is not in computing individual trajectories. For any fixed starting value, the process is mechanically defined. The difficulty is in proving a universal statement about all positive integers.

The map combines two behaviors. Even integers decrease immediately by division by 2. Odd integers are mapped to $3n+1$, which is even and larger than n for all positive odd $n > 1$. A trajectory can therefore rise before it falls. Some starting values produce short trajectories. Others produce long trajectories with large intermediate values before eventually descending to 1.

This mixture of local growth and repeated division makes the problem resistant to direct monotonic arguments. A proof cannot rely only on the observation that many tested values eventually decrease. It must account for every possible trajectory, including those that may rise for long intervals or reach large intermediate values.

The conjecture can also be studied through related maps, accelerated variants, parity sequences, modular structure, stopping times, and probabilistic or density-based arguments. These perspectives have produced partial results and useful reformulations, but they have not produced a proof of the full conjecture [1, 2].

1.3 Computation and Its Limits

Computation has played a significant role in the study of the $3n + 1$ problem. For a finite range of starting values, one can evaluate each trajectory and record whether it reaches 1. Such computation can establish that the conjecture holds for all starting values in the checked range. It cannot establish that the conjecture holds for all positive integers.

This distinction is important. A finite verification, no matter how large, is not a proof of the conjecture. The conjecture is a universal statement over an infinite set. A computation that checks values up to a bound B proves only the bounded statement:

$$\text{For every positive integer } n \leq B, \text{ the trajectory of } n \text{ reaches } 1.$$

It does not prove the unbounded statement:

$$\text{For every positive integer } n, \text{ the trajectory of } n \text{ reaches } 1.$$

Computational exploration remains useful, but its role must be stated carefully. It can identify record trajectories, measure stopping times, locate high peak values, test implementations, produce datasets, and support independent verification of finite ranges. It can also help expose errors in assumptions about how trajectories behave. It does not replace proof.

For a public computational system, this distinction should be visible in the design of the system itself. The system should not present numerical progress as mathematical resolution. It should present numerical progress as finite verified computation.

1.4 Purpose and Scope of This Report

This report describes Collatz Engine as a persistent public computational observatory for the $3n+1$ problem.

Collatz Engine is a computational system for sequentially evaluating the Collatz map over positive integers and recording the resulting trajectories, stopping times, peak values, and record events. The system is designed to continue computation over time, preserve state across interruptions, and expose the current state of the computation through a public dashboard.

The purpose of this report is not to present a proof of the Collatz conjecture. It does not claim that the conjecture has been solved, weakened, or reduced to a finite computation. The purpose is to document the system architecture, the computation being performed, the stored quantities, the persistence model, and the public reporting layer.

The report focuses on four questions.

First, what exactly does the system compute?

Second, how does the system preserve progress across redeployments, crashes, refreshes, and downtime?

Third, what data is stored, and how can the stored data support inspection or independent verification?

Fourth, how should the public dashboard report finite computational progress without overstating its mathematical significance?

These questions are practical. A long-running public computation needs clear definitions, persistent state, reproducible methods, and careful language. Without these, the system may produce numbers that look precise but are difficult to audit.

The initial version of Collatz Engine evaluates starting integers sequentially. For each starting value, it applies the Collatz map until the trajectory reaches the known cycle containing 1. The system records computed quantities such as trajectory length, peak value, and record-breaking events. Engine-level state, including current number, total checked values, runtime accounting, and status, is stored in a database rather than in browser memory. This allows the system to resume from the last stored state after an interruption.

The public interface is part of the system being documented. It is not only a visual layer. It reports the state of a running computation. For that reason, the interface must distinguish between live status, historical records, computed observations, and mathematical claims. The dashboard may show the current integer being evaluated, the number of values checked, the longest trajectory observed, the highest peak encountered, and throughput over time. These values are empirical records from finite computation.

The scope of this report is limited to the design and operation of the current computational system. It does not attempt to survey the full mathematical literature on the Collatz conjecture. It also does not attempt to compare Collatz Engine with all previous computational verification efforts.

Where historical or mathematical context is needed, the report uses it only to clarify the system's purpose and limitations.

The intended contribution of this report is documentation. It defines the computation, records the architecture, and establishes the language by which the system should be interpreted. Future reports may address larger verified ranges, distributed computation, public datasets, independent verifier tooling, or statistical summaries of observed trajectories.

2 System Overview

2.1 Definition of Collatz Engine

Collatz Engine is a persistent computational system for evaluating the Collatz map over positive integers and reporting finite computational results through a public interface. The system is designed to maintain progress over time, record computed results, and expose selected state variables in a way that can be inspected by external readers.

The term *engine* is used in this report to refer to the combined computation, persistence, and reporting system. It does not refer only to the visual dashboard. The engine consists of a worker process that evaluates starting integers, a database layer that stores state and results, and a public interface that presents current and historical computation status.

The current implementation evaluates starting integers sequentially. Beginning from a stored current value, the worker evaluates one or more integers, records the results, updates engine-level state, and advances the stored current value. If the system is interrupted, the next run is expected to resume from the last persisted state rather than from an in-memory value.

The system is public-facing, but it is not presented as a proof system. It records finite computation. A verified range of checked integers may support bounded claims about that range. It does not support an unbounded claim about all positive integers.

2.2 Sequential Evaluation Model

The baseline computation model is sequential. The engine evaluates positive integers in increasing order, beginning at 1 and advancing upward. For each starting value n, the engine applies the Collatz map until the trajectory reaches the known cycle containing 1. The result for that starting value is then recorded.

This sequential model has several advantages for a first public implementation.

First, it provides a simple interpretation of progress. If the stored current value is N, and the engine has not skipped values, then the computation can be interpreted as having covered the positive integers below N, subject to the integrity of the stored records and update logic.

Second, it reduces ambiguity in public reporting. The dashboard can report the current integer under evaluation and the total number of checked integers without requiring the reader to reconstruct a distributed work schedule.

Third, it creates a straightforward basis for audit. A sequential range can be independently recomputed by another implementation. Discrepancies can then be investigated by comparing starting values, trajectory lengths, peak values, and termination status.

The sequential model is not the only possible model. A future implementation may distribute ranges across multiple workers. Such a system would require additional coordination, range assignment, conflict prevention, and verification logic. The initial system avoids these requirements by using a single forward-moving computation model.

2.3 Core System Components

The system can be described in terms of four primary components.

The first component is the computation worker. The worker is responsible for selecting starting integers, evaluating trajectories, calculating per-integer quantities, and preparing updates to the stored engine state. In the current architecture, this worker is separate from the browser session. This distinction matters because the computation should not depend on a user keeping a page open.

The second component is the persistence layer. The persistence layer stores engine state, computed results, record events, and operational timestamps. Engine state includes values such as the current integer, the total number of checked integers, the longest trajectory observed, the highest peak observed, engine status, and update timestamps. The database is the source of truth for progress.

The third component is the public dashboard. The dashboard reads from the persistence layer and presents selected state variables to the public. It may show current number, total checked values, runtime, throughput, longest trajectory, highest peak, and recent record events. The dashboard should be treated as a reporting surface, not as the computation authority.

The fourth component is the visualization layer. This layer renders trajectories, heatmaps, descent events, record histories, and other views derived from stored or computed data. These visualizations are useful for inspection and communication, but they do not change the mathematical status of the computation.

A simplified architecture is shown in Figure 1.

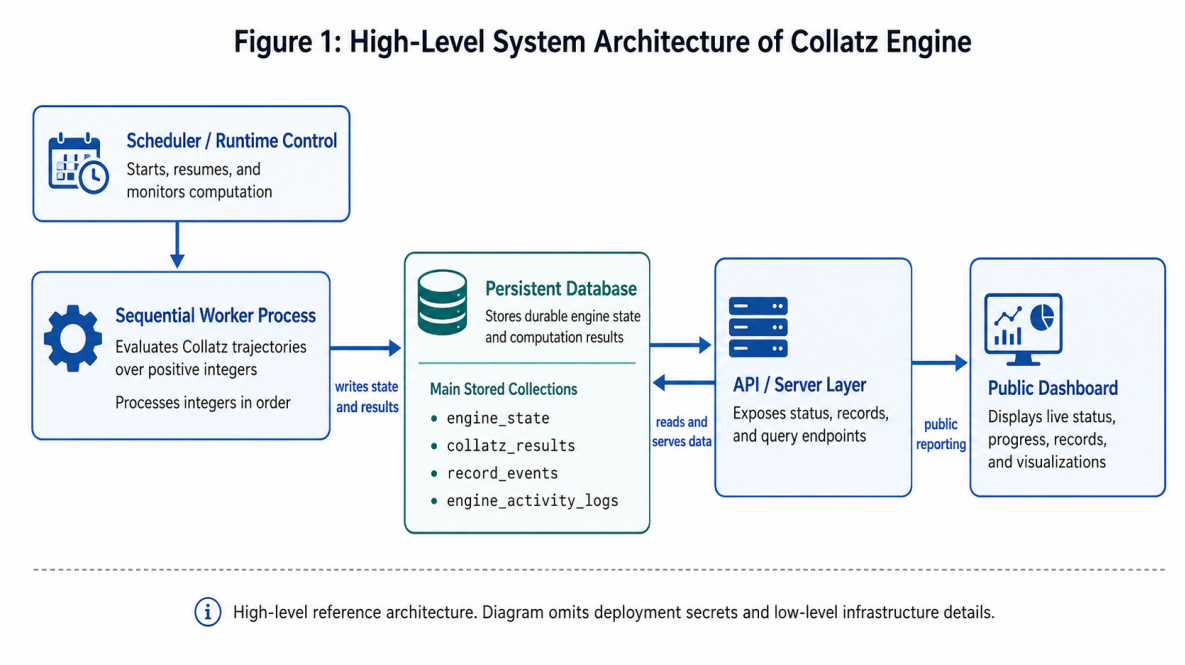

Figure 1: High-level system architecture of Collatz Engine. The diagram shows the relationship between scheduler and runtime control, the sequential worker process, the persistent database, the API or server layer, and the public dashboard.

2.4 Public Observability Requirements

A public computational system should expose enough information for readers to understand what is being computed, what has already been checked, and how progress is being recorded. For Collatz Engine, public observability has three purposes.

The first purpose is status reporting. A reader should be able to see whether the engine is running, paused, delayed, or unavailable. This status should come from stored state and timestamps rather than from a browser-only timer.

The second purpose is computational interpretation. A reader should be able to distinguish between values that describe finite computation and statements about the conjecture itself. For example, a dashboard value such as "total checked" refers to a finite count. It does not imply that the conjecture has been proved.

The third purpose is operational transparency. A reader should be able to inspect enough system behavior to understand whether the engine is advancing, whether records are changing, and whether the reported runtime is derived from persistent data. Public logs, record feeds, timestamps, and historical charts can support this goal.

The dashboard should therefore avoid ambiguous language. It should not describe the engine as solving the Collatz conjecture. It should describe the engine as evaluating, recording, and visualizing finite trajectories under the Collatz map.

2.5 State Persistence as a Design Requirement

Persistence is a central requirement of the system. A computation that runs only in browser memory is not sufficient for a public long-running engine. Browser sessions can close, refresh, crash, or become inactive. Deployment environments can restart. Network requests can fail. If progress is stored only in memory, then the reported state can become unreliable.

For this reason, Collatz Engine uses database-backed state. The current integer, aggregate counters, record values, engine status, and relevant timestamps are stored outside the browser. The computation should be able to resume from this stored state after interruption.

Persistent timestamps are also required for runtime accounting. A browser timer can show how long a page has been open, but it cannot reliably show how long the engine has been operating across redeployments or downtime. Runtime should be computed from stored timestamps and status transitions, not from client-side elapsed time alone.

This design separates the public display from the computation record. The browser may render a live clock or update visible values, but the durable state remains in the database. The database state is the reference point for restart behavior, public reporting, and future audit.

2.6 System Boundaries and Non-Goals

The current system has defined boundaries.

It evaluates finite ranges of positive integers under the Collatz map. It records observed trajectory behavior for those starting values. It reports current computation state and selected historical records. It may provide visualizations that help readers inspect trajectory behavior.

The system does not prove the Collatz conjecture. It does not claim to identify a counterexample. It does not claim that observed finite behavior establishes the behavior of all positive integers. It does not replace mathematical proof with computation.

The system also does not attempt, in its initial form, to implement a fully distributed verification network. Distributed computation is a possible future direction, but it introduces additional requirements: worker coordination, range assignment, duplicate prevention, result reconciliation, and independent validation. These concerns are outside the first system boundary.

The public interface should reflect these boundaries. It should report finite computation accurately and avoid language that assigns mathematical significance beyond what the stored data supports.

2.7 Interpretation of the System

Collatz Engine should be interpreted as computational infrastructure for observing finite behavior of the Collatz map. Its value depends on precise computation, persistent state, transparent reporting, and careful separation between empirical records and mathematical claims.

The system can support bounded verification statements over checked ranges. It can produce records of long trajectories, high peaks, descent patterns, and operational progress. It can also provide a public record of how a long-running computation advances over time.

These outputs are useful only if the system remains clear about what they mean. A finite computation can be correct, reproducible, and publicly visible without being a proof of the conjecture. This report adopts that distinction as a design constraint.

3 Mathematical Definitions and Computed Quantities

3.1 Purpose of the Definitions

This section defines the mathematical quantities used by Collatz Engine. The definitions are included to avoid ambiguity in later sections of the report. A public computation can report large numerical values, but those values are not meaningful unless the reported quantities are defined consistently.

The definitions in this section are standard or directly derived from the standard Collatz map. Where naming conventions vary in the literature, this report states the convention used by the system.

3.2 Iteration Function

Let $\mathbb{Z}_{>0}$ denote the set of positive integers. The Collatz map is the function

$$T : \mathbb{Z}_{>0} \to \mathbb{Z}_{>0}$$

defined by

$$T(n) = \begin{cases} n/2, & \text{if } n \equiv 0 \pmod 2, \\ 3n+1, & \text{if } n \equiv 1 \pmod 2. \end{cases}$$

The k-fold iterate of T is denoted by $T^k(n)$, where

$$T^0(n) = n$$

and

$$T^{k+1}(n) = T(T^k(n)).$$

Under this notation, a computation beginning at n produces the values

$$T^0(n),\ T^1(n),\ T^2(n),\ T^3(n),\ \ldots$$

The Collatz conjecture states that for every $n \in \mathbb{Z}_{>0}$, there exists a nonnegative integer k such that

$$T^k(n) = 1.$$

This report does not assume the conjecture as a proven statement. The engine evaluates finite cases and records whether the computed trajectory reaches the known cycle containing 1.

3.3 Trajectory

For a starting value n, the trajectory of n is the finite or infinite sequence obtained by repeated application of the Collatz map:

$$\mathcal{T}(n) = \left(T^0(n), T^1(n), T^2(n), \ldots\right).$$

If the trajectory reaches 1, then the computed finite trajectory is recorded up to the first occurrence of 1. In that case, the recorded trajectory is

$$\mathcal{T}_1(n) = \left(T^0(n), T^1(n), \ldots, T^{\sigma_\infty(n)}(n)\right),$$

where $\sigma_\infty(n)$ is the total stopping time defined below.

For example, the trajectory beginning at $n = 7$, recorded up to the first occurrence of 1, is

$$7,\ 22,\ 11,\ 34,\ 17,\ 52,\ 26,\ 13,\ 40,\ 20,\ 10,\ 5,\ 16,\ 8,\ 4,\ 2,\ 1.$$

The engine does not need to store every full trajectory permanently in order to maintain aggregate records. It may compute a trajectory, derive quantities such as stopping time and peak value, and store only the result fields required by the data model. Full trajectory storage is a separate design choice and is discussed later in the report.

3.4 Total Stopping Time

For a starting value n, the total stopping time is the least nonnegative integer k such that the trajectory reaches 1:

$$\sigma_\infty(n) = \min\{k \geq 0 : T^k(n) = 1\}.$$

This value is defined only when such a k exists. If the trajectory of n has not been shown to reach 1, then the total stopping time is not established for that n.

For example, for $n = 7$, the first occurrence of 1 is reached after 16 applications of T. Therefore,

$$\sigma_\infty(7) = 16.$$

This report uses total stopping time to mean the number of Collatz map applications required to reach 1. This is distinct from a trajectory length that counts the starting value as an element of the recorded sequence.

3.5 Trajectory Length

The recorded trajectory length is the number of values in the trajectory from the starting value through the first occurrence of 1. Under the convention used in this report,

$$L(n) = \sigma_\infty(n) + 1.$$

The additional 1 accounts for the initial value $T^0(n) = n$.

For $n = 7$, the total stopping time is 16, so the recorded trajectory length is

$$L(7) = 17.$$

This distinction matters in public reporting. If one implementation reports the number of transitions and another reports the number of listed values, the two numbers will differ by 1. Collatz Engine should state which convention is used wherever trajectory length or stopping time is displayed.

3.6 Peak Value

The peak value of a trajectory is the maximum value reached before the first occurrence of 1. For a starting value n whose trajectory reaches 1, the peak value is defined as

$$P(n) = \max_{0 \le j \le \sigma_\infty(n)} T^j(n).$$

For $n = 7$, the recorded trajectory is

$$7,\ 22,\ 11,\ 34,\ 17,\ 52,\ 26,\ 13,\ 40,\ 20,\ 10,\ 5,\ 16,\ 8,\ 4,\ 2,\ 1.$$

The largest value in this trajectory is 52, so

$$P(7) = 52.$$

Peak values are useful because trajectories do not necessarily decrease monotonically. A starting value may rise substantially before descending. The peak value records the largest intermediate integer encountered during the computation for a given starting value.

3.7 Known Cycle Containing 1

Once a trajectory reaches 1, repeated application of the Collatz map gives

$$1 \mapsto 4 \mapsto 2 \mapsto 1.$$

This is the known cycle containing 1. For the purposes of the engine, reaching 1 is treated as the termination condition for a computed trajectory. The engine does not need to continue storing the repeated cycle after the first occurrence of 1.

This termination convention is operational. It does not assert that no other cycles exist for positive integers. It only states that, for a trajectory that reaches 1, the subsequent behavior is already determined by the known cycle.

3.8 Record Events

A record event occurs when a newly evaluated starting value exceeds a previously stored aggregate record. Collatz Engine tracks at least two classes of record events.

The first class is a longest-trajectory record. If a starting value n produces a trajectory length $L(n)$ greater than the previous maximum recorded trajectory length, then n establishes a new longest-trajectory record.

The second class is a highest-peak record. If a starting value n produces a peak value $P(n)$ greater than the previous maximum recorded peak value, then n establishes a new highest-peak record.

These records are empirical records within the checked finite range. If the engine has evaluated all starting values up to a bound B, then the records are meaningful over that checked range. They should not be interpreted as global records over all positive integers unless such a claim is independently established.

3.9 Per-Integer Result

For each starting value n, the engine may produce a per-integer result containing the following fields:

- the starting value n;
- the total stopping time $\sigma_\infty(n)$, when the trajectory reaches 1;

- the recorded trajectory length $L(n)$;
- the peak value $P(n)$;
- the termination status;
- the timestamp at which the result was computed or stored.

Additional fields may be included for operational or analytical purposes. These may include worker identifier, batch identifier, computation duration, parity summary, descent count, or serialized trajectory data. Such fields are implementation details unless they are used in public reporting or independent verification.

The minimum useful result is one that records the starting value, whether the trajectory reached 1, the total stopping time or trajectory length, and the peak value. These quantities are sufficient to support basic dashboard records and bounded recomputation checks.

3.10 Aggregate Engine State

In addition to per-integer results, the engine maintains aggregate state. Aggregate state describes the current progress and summary records of the running computation. It may include:

- the next starting value to be evaluated;
- the total number of starting values checked;
- the largest checked starting value;
- the current longest trajectory length;
- the starting value that produced the current longest trajectory;
- the current highest peak value;
- the starting value that produced the current highest peak;
- engine status;
- runtime accounting fields;
- timestamps for creation, update, start, pause, and resume events.

Aggregate state should be stored persistently. If these values exist only in memory, then the engine cannot reliably resume after interruption and cannot provide a durable public record of progress.

3.11 Finite Verification Statement

The engine's computation supports finite verification statements. If the engine has evaluated every starting value in the interval

$$1 \leq n \leq B$$

and each evaluated trajectory has reached 1, then the stored computation supports the bounded statement:

$$\forall n \in \mathbb{Z}_{>0},\ n \leq B \Rightarrow \exists k \geq 0 \text{ such that } T^k(n) = 1.$$

This statement is finite. It should not be restated as a proof of the Collatz conjecture. The conjecture requires the corresponding unbounded statement for all positive integers:

$$\forall n \in \mathbb{Z}_{>0},\ \exists k \geq 0 \text{ such that } T^k(n) = 1.$$

The distinction between these two statements is a core reporting requirement for Collatz Engine. The dashboard, documentation, and public summaries should preserve it.

4 Computation Method

4.1 Purpose of the Computation Method

This section describes how Collatz Engine evaluates starting integers and derives the quantities defined in Section 3. The method is intentionally simple in the initial implementation. The engine evaluates starting values in increasing order, computes each trajectory under the Collatz map, records the result, and advances persistent state.

The computation method has three design requirements.

First, the computation must be deterministic. Given the same starting value, the same implementation of the Collatz map should produce the same trajectory, total stopping time, trajectory length, and peak value.

Second, progress must be durable. The engine should not depend on browser memory or a user session to determine the next starting value.

Third, the method must support inspection. A stored result should contain enough information for another implementation to recompute the same starting value and compare the result.

4.2 Starting Integer Selection

The engine uses a sequential selection rule. Let c denote the next starting integer stored in persistent engine state. The worker selects c, computes the result for c, stores the result, and then advances the stored next value to $c + 1$.

In batch execution, the worker may reserve or process an interval

$$[c, c + b - 1],$$

where b is the batch size. After the batch is completed and persisted, the next stored value becomes

$$c + b.$$

Sequential selection makes the checked range easier to interpret. If the engine has evaluated every integer from 1 through B, then the public system may report that the finite range $1 \leq n \leq B$ has been checked, subject to the integrity of the stored records and state transitions.

The initial system should avoid claiming coverage for values that were selected but not successfully persisted. A starting value should be counted as checked only after its result has been written to storage or otherwise included in a durable transaction.

4.3 Trajectory Evaluation

For each selected starting value n, the worker initializes the current trajectory value as

$$x_0 = n.$$

At each step j, the worker computes the next value using the Collatz map:

$$x_{j+1} = T(x_j).$$

The computation continues until the current value reaches 1. The worker then records the total number of map applications required to reach 1, the number of values in the recorded trajectory, and the maximum value encountered.

The following recurrence describes the computed trajectory:

$$x_j = T^j(n)$$

with

$$x_0 = n.$$

The termination condition for the computed trajectory is

$$x_j = 1.$$

When this condition is first met, the worker stops the per-integer computation.

4.4 Operational Pseudocode

The per-integer computation can be expressed in operational form as follows:

```
function analyze(n):
    current = n
    steps = 0
    peak = n

    while current != 1:
        if current is even:
```

```
            current = current / 2
        else:
            current = 3 * current + 1

        steps = steps + 1

        if current > peak:
            peak = current

    trajectory_length = steps + 1

    return {
        start: n,
        total_stopping_time: steps,
        trajectory_length: trajectory_length,
        peak_value: peak,
        reached_one: true
    }
```

This pseudocode counts `steps` as the number of applications of the Collatz map. The recorded trajectory length is `steps + 1`, because it includes the starting value.

The pseudocode also treats reaching 1 as the termination condition. It does not continue through the repeated cycle $1 \mapsto 4 \mapsto 2 \mapsto 1$.

4.5 Per-Integer Result Generation

After a starting value has been evaluated, the worker creates a per-integer result. At minimum, the result should include:

- the starting value;
- the total stopping time;
- the recorded trajectory length;
- the peak value;
- the termination status;
- the time at which the result was stored.

The termination status should explicitly distinguish a completed trajectory from an interrupted or failed computation. In the current system boundary, a completed result means that the trajectory

reached 1 under the implemented map and the result was persisted.

The system may also store auxiliary values, such as compute duration, batch identifier, worker identifier, or a compact representation of the trajectory. These values are not required for the mathematical definition of the result, but they may be useful for audit, debugging, or future distributed execution.

4.6 Aggregate Record Updates

After each per-integer result is generated, the engine compares the result against stored aggregate records.

If the new trajectory length exceeds the stored longest trajectory length, the engine updates the longest-trajectory record. The updated record should include both the new length and the starting value that produced it.

If the new peak value exceeds the stored highest peak value, the engine updates the highest-peak record. The updated record should include both the new peak and the starting value that produced it.

For a starting value n, a longest-trajectory update occurs when

$$L(n) > L_{\max},$$

where $L_{\max}$ is the previously stored maximum trajectory length.

A highest-peak update occurs when

$$P(n) > P_{\max},$$

where $P_{\max}$ is the previously stored maximum peak value.

These updates should be persisted with timestamps. If the system maintains a record-event feed, then each record update should also produce an event entry that can be displayed or inspected later.

4.7 Batch Execution

The worker may process multiple starting values in a single batch. Batch execution reduces database write overhead and allows aggregate state to be updated after a group of computations rather than after every single starting value.

A batch can be described as an ordered interval of starting values:

$$n = c,\ c+1,\ c+2,\ \ldots,\ c+b-1.$$

For each value in the interval, the worker computes a result. The worker then writes the per-integer results and updates aggregate state. If the implementation accumulates batch results in memory before writing them, it should flush the accumulated results at defined intervals or at the end of the batch.

Batch execution introduces a persistence requirement. The system should not advance the stored next starting value beyond the durable results unless the corresponding results have been stored or the state update is otherwise atomic. If the next starting value advances before results are durable, an interruption could create a gap in the checked range.

4.8 Handling Intermediate Values

Collatz trajectories may contain intermediate values larger than the starting value. Therefore, the implementation must use an integer representation capable of holding values encountered during computation.

For small checked ranges, ordinary fixed-width integers may be sufficient. For larger ranges, fixed-width integer overflow becomes a correctness risk. If a computed intermediate value exceeds the numeric type's maximum representable value, the resulting trajectory may be wrong even if the program continues to run.

A robust implementation should use arbitrary-precision integer arithmetic or otherwise enforce bounds that prevent overflow. In JavaScript and TypeScript environments, this generally means using `BigInt` for values that may exceed the safe integer range of the standard `number` type.

This requirement applies to both the starting value and intermediate trajectory values. It also applies to stored peak values if the database field must preserve exact integer values.

4.9 Termination and Failure Modes

For a computed starting value, normal termination occurs when the trajectory reaches 1. The worker then records the result as completed.

Other outcomes should be treated as operational failure modes, not as mathematical conclusions. Examples include:

- process interruption before completion;
- database write failure;
- timeout;

- numeric overflow;
- invalid input state;
- duplicate batch execution;
- inconsistent stored state.

None of these conditions should be reported as evidence for or against the Collatz conjecture. They are system conditions. The engine should log them, avoid advancing durable progress incorrectly, and resume from a known stored state.

If the system introduces a maximum iteration limit for safety, then reaching that limit must be reported as an incomplete computation. It should not be reported as a non-converging trajectory unless a mathematical counterexample has actually been established and independently verified.

4.10 Determinism and Recomputability

The Collatz map is deterministic. For a given starting value n, the trajectory under the standard map is uniquely determined. This property allows independent recomputation.

A stored result for n can be checked by another implementation that computes the same map and compares:

- total stopping time;
- trajectory length;
- peak value;
- termination status.

If these values differ, then at least one implementation or stored record is inconsistent. The discrepancy may be caused by an arithmetic error, an off-by-one convention, overflow, a storage problem, or a different definition of the computed quantity.

For this reason, Collatz Engine should clearly define whether public values refer to total stopping time or trajectory length. It should also state whether peak values include the starting value. In this report, peak values include the starting value, and trajectory length includes the starting value and the first occurrence of 1.

4.11 Summary

The initial computation method is sequential, deterministic, and database-backed. The worker selects starting integers from persistent state, evaluates each trajectory under the Collatz map, records per-integer results, updates aggregate records, and advances the stored next value only as durable progress is made.

This method does not prove the Collatz conjecture. It supports finite computation over checked ranges and produces records that can be inspected, recomputed, and used for public reporting.

5 Persistent Engine State

5.1 Purpose of Persistent State

Persistent state is required for Collatz Engine because the computation is intended to continue across browser sessions, deployments, process restarts, and periods of downtime. A public long-running computation cannot depend on a value stored only in memory. If the current starting integer, aggregate records, or runtime counters are held only by a browser or temporary process, then the system cannot reliably resume from the last known point after interruption.

In this report, persistent state refers to engine-level values stored in a durable database. These values describe the progress and operational condition of the computation. They are distinct from per-integer results, which store the outcome of individual trajectory evaluations.

Persistent state serves four functions.

First, it identifies the next starting integer to be evaluated.

Second, it records aggregate progress, including the total number of checked values and the highest contiguous range covered by the computation.

Third, it stores aggregate records, such as the longest trajectory observed and the highest peak encountered.

Fourth, it records operational timing and status information used by the public dashboard.

5.2 Database as Source of Truth

The database should be treated as the source of truth for engine progress. The browser may display values, animate counters, or poll an API endpoint, but it should not define durable progress. Similarly, a worker process may hold temporary values during a batch, but those values should not be considered public state until they have been persisted.

This distinction is important because public reporting can otherwise become inconsistent. A browser timer may continue to run after computation has stopped. A worker may compute a result but fail before writing it. A deployment may restart while values exist only in process memory. In each case, the durable state must come from the database rather than the transient execution environment.

The engine should therefore follow this principle:

Public progress is the progress stored in persistent state.

This principle does not prevent local buffering or batch processing. It only means that buffered

work should not be reported as completed until it has been made durable.

5.3 Engine State Variables

The persistent engine state should contain a compact set of variables sufficient to resume computation and report public progress. The exact database schema may evolve, but the following fields describe the core state model:

- `current_number`: the next starting integer to be evaluated;
- `total_checked`: the total number of starting integers completed and persisted;
- `highest_checked`: the largest starting integer included in the completed checked range;
- `longest_trajectory_length`: the largest recorded trajectory length observed so far;
- `longest_trajectory_start`: the starting integer that produced the current longest trajectory;
- `highest_peak_value`: the largest intermediate value observed so far;
- `highest_peak_start`: the starting integer that produced the current highest peak;
- `engine_status`: the current operational status of the engine;
- `started_at`: the timestamp associated with the beginning of the current or most recent run period;
- `last_updated_at`: the timestamp of the most recent durable state update;
- `last_heartbeat_at`: the timestamp of the most recent worker heartbeat, if heartbeat reporting is used.

These fields are engine-level fields. They summarize progress. They do not replace the per-integer results table, which stores individual computation outcomes.

5.4 State Transition Model

The engine should have a small set of operational states. A typical state model may include:

- `running`: the worker is actively evaluating starting integers or is expected to be active;
- `paused`: the engine has been intentionally stopped or suspended;
- `stalled`: no recent heartbeat or state update has been observed within the expected interval;
- `error`: an operational failure has been detected;
- `initializing`: the engine is preparing to begin or resume computation.

These states are operational. They should not be interpreted as mathematical outcomes. For example, an `error` state means that the system encountered a technical problem. It does not indicate anything about the Collatz conjecture.

State transitions should be written to durable storage. If the engine moves from `running` to `paused`, that transition should be recorded. If the worker resumes, the resume event should be reflected in persistent state. This allows the dashboard to report status based on stored evidence rather than local assumptions.

5.5 Resume Behavior After Interruption

Resume behavior is one of the main reasons persistent state is required. When the worker starts, it should read the stored engine state before selecting a starting value. The stored `current_number` or equivalent field determines where computation resumes.

A safe resume sequence can be described as follows:

1. read the current persistent engine state;
2. verify that the engine is allowed to run;
3. select the next starting value or batch range from persistent state;
4. compute the selected value or range;
5. persist per-integer results;
6. update aggregate records and progress counters;
7. advance the stored current number;
8. write the update timestamp and heartbeat.

The important rule is that the stored current number should not be advanced in a way that creates an unverified gap. If the worker computes a batch but fails before writing the results, the system should be able to retry the same batch or otherwise reconcile the incomplete work.

For a sequential engine, gaps are especially important. If the public system claims that all starting integers up to B have been checked, then the stored results and state transitions should support that claim. Advancing state before durable result storage weakens that interpretation.

5.6 Timestamp-Based Runtime Accounting

Runtime should be derived from persistent timestamps rather than browser-only timers. A browser timer measures the duration of a page session. It does not measure the duration of engine operation across refreshes, closed tabs, deployments, restarts, or downtime.

A database-backed runtime model may use timestamps such as:

- `created_at`: when the engine state record was created;
- `started_at`: when computation began or resumed;
- `paused_at`: when computation was intentionally paused;
- `last_updated_at`: when durable progress was last written;
- `last_heartbeat_at`: when the worker last signaled activity.

For public reporting, the dashboard should distinguish between elapsed wall-clock time and active computation time. Wall-clock time may include periods when the engine was offline. Active computation time should include only periods when the engine was running or expected to be running.

If exact active runtime is required, the system should record run intervals. A run interval has a start timestamp and an end timestamp. The total active runtime is then the sum of completed run intervals plus the duration of the current active interval, if the engine is running.

5.7 Heartbeat and Staleness Detection

A heartbeat is a periodic update written by the worker to indicate that it is alive. Heartbeat timestamps are useful for detecting stale public state. If the dashboard shows a status of `running`, but the most recent heartbeat is too old, the system may report the engine as stale or stalled.

A heartbeat should not be confused with completed computation. It indicates worker activity, not necessarily durable progress. For example, a worker could be alive but unable to write results. For that reason, heartbeat information should be considered alongside `last_updated_at`, total checked values, and error logs.

A simple staleness rule may be expressed as follows. Let t_{now} be the current server time, and let $t_{\text{heartbeat}}$ be the stored heartbeat time. If

$$t_{\text{now}} - t_{\text{heartbeat}} > \Delta,$$

where Δ is the allowed heartbeat interval, then the engine state may be considered stale.

The value of Δ is an operational parameter. It should be chosen based on the worker schedule, expected batch duration, and deployment environment.

5.8 Concurrency and Locking

A sequential engine should prevent more than one worker from advancing the same state at the same time. If two workers read the same current number and both process overlapping ranges, the system may produce duplicates, inconsistent counters, or conflicting record updates.

A basic concurrency control mechanism is therefore required if there is any possibility of overlapping worker execution. This may be implemented using a database lock, a lease record, an advisory lock, or an atomic update condition. The mechanism should ensure that only one active worker can claim a given range of starting integers.

A lock or lease should include an expiration condition. Without expiration, a worker crash could leave the engine permanently locked. With expiration, the system can recover if the lock holder stops updating its heartbeat.

The lock should be interpreted operationally. It does not affect the mathematical computation. It protects the integrity of state transitions and prevents duplicate or skipped work.

5.9 Atomicity of Progress Updates

Progress updates should be atomic where possible. If a batch of results is written and aggregate state is advanced, these operations should either succeed together or be recoverable if partially completed.

The risk can be illustrated with a batch beginning at c and containing b starting values. If the engine advances `current_number` to $c + b$ before the results for the batch are durable, then a crash could cause the engine to resume after the batch without having stored the corresponding results.

A safer approach is to persist results first and advance the engine state only after the results are durable. If the database supports transactions, the result writes and state update can be grouped. If full transactional grouping is not available in the deployed path, the system should use idempotent writes and reconciliation logic to avoid gaps.

Idempotent writes allow the same starting value to be written more than once without creating duplicate logical results. This can be enforced through unique constraints on the starting value or through conflict-handling logic during insert.

5.10 Public State Versus Internal State

Not all state fields need to be public. The dashboard should expose fields that help readers understand the computation. Internal fields may be needed for worker coordination, debugging, retry behavior, or operational diagnostics.

Public state may include:

- current number;
- total checked;
- largest checked value;
- longest trajectory record;
- highest peak record;
- engine status;
- last updated time;
- throughput estimate.

Internal state may include:

- lock owner;
- lock expiration time;
- batch identifier;
- worker identifier;
- retry count;
- last error message;
- diagnostic metadata.

The distinction protects both clarity and security. Public reporting should be sufficient for interpretation, but it should not expose sensitive operational details that could interfere with the running system.

5.11 Failure Recovery

Failure recovery should be based on persistent evidence. When the system restarts after interruption, it should determine the next action from stored state, not from assumptions about what the previous worker may have completed.

Typical recovery actions include:

- resume from the stored current number;
- retry an incomplete batch;
- release an expired worker lock;

- mark stale state as stalled;
- reconcile result records against aggregate counters;
- recompute aggregate records if inconsistency is detected.

Recovery logic should prefer correctness over speed. It is better for the engine to repeat a range that can be deduplicated than to skip a range and falsely report contiguous verification.

Failure recovery should also preserve public interpretation. A technical interruption is not evidence of unusual mathematical behavior. It should be logged as an operational event.

5.12 Summary

Persistent engine state is the mechanism by which Collatz Engine maintains continuity over time. It stores the next starting value, aggregate progress, record values, runtime timestamps, and operational status. It allows the worker to resume after interruption and allows the dashboard to report values derived from durable data.

The database, not the browser, should be treated as the source of truth. This design requirement supports reproducibility, public observability, and auditability. It also prevents the system from presenting transient or incomplete computation as verified progress.

6 Data Model and Storage Design

6.1 Purpose of the Data Model

The data model for Collatz Engine has two responsibilities. It must preserve the state required to continue computation, and it must store the results needed to inspect completed work. These responsibilities are related, but they should not be treated as the same table or the same logical record.

Engine state describes where the computation is and what aggregate records have been observed so far. Per-integer results describe what was computed for individual starting values. Record events describe changes in aggregate records. Activity logs describe operational behavior of the system.

Separating these concerns makes the system easier to inspect. If the dashboard reports a total checked count, that count should be traceable to persistent state and stored results. If the engine reports a new longest trajectory, that event should be linked to the starting value and result that produced it. If the worker stops or restarts, the operational record should be distinguishable from the mathematical computation.

The schema described in this section is a reference design. Field names may differ in the implementation, but the logical responsibilities should remain distinct.

6.2 Primary Tables

The current system can be represented by four primary table groups:

- `engine_state`, which stores current progress and aggregate state;
- `collatz_results`, which stores per-integer computation results;
- `record_events`, which stores new longest-trajectory and highest-peak events;
- `engine_activity_logs`, which stores operational events, status changes, and diagnostic messages.

Additional tables may be introduced for distributed workers, exported datasets, independent verification runs, or dashboard analytics. These extensions should preserve the distinction between computation results and operational metadata.

6.3 Engine State Table

The `engine_state` table stores the durable state of the running computation. In a single-engine deployment, this table may contain one active row. In a future multi-engine or multi-environment deployment, it may include an engine identifier to distinguish separate computation streams.

A reference schema is shown in Table 1.

Table 1: Reference fields for persistent engine state.

Field	Type	Purpose
id	UUID or integer	Unique engine state record identifier.
current_number	integer or numeric text	Next starting integer to evaluate.
total_checked	integer	Number of completed and persisted starting values.
highest_checked	integer or numeric text	Largest starting value included in the checked range.
longest_trajectory_length	integer	Longest recorded trajectory length observed so far.
longest_trajectory_start	integer or numeric text	Starting value that produced the current longest trajectory.
highest_peak_value	numeric text	Highest intermediate value observed so far.
highest_peak_start	integer or numeric text	Starting value that produced the current highest peak.
engine_status	text	Operational state, such as running, paused, stalled, or error.
started_at	timestamp	Timestamp for the start or most recent resume event.
last_updated_at	timestamp	Timestamp for the most recent durable progress update.
last_heartbeat_at	timestamp	Timestamp for the most recent worker heartbeat.
created_at	timestamp	Record creation timestamp.
updated_at	timestamp	Record update timestamp.

The field `current_number` should refer to the next starting value to be evaluated, not the last completed value. This convention avoids ambiguity. If the engine has completed all values through B, then `current_number` should be $B + 1$, and `highest_checked` should be B.

Large integer fields require care. JavaScript's standard `number` type cannot safely represent all large integers. If starting values or peak values may exceed the safe integer range, the implementation should use arbitrary-precision arithmetic and store large values in a representation that preserves exactness. In a database, this may require a high-precision numeric type or a text field with validation.

6.4 Results Table

The `collatz_results` table stores the result of evaluating a starting value. Each logical starting value should have at most one completed result under a given computation definition. A unique constraint on `start_value` can help prevent duplicate logical records.

A reference schema is shown in Table 2.

Table 2: Reference fields for per-integer Collatz results.

Field	Type	Purpose
`id`	UUID or integer	Unique result identifier.
`start_value`	integer or numeric text	Starting integer n.
`total_stopping_time`	integer	Number of map applications required to reach 1.
`trajectory_length`	integer	Number of recorded values from n through first 1.
`peak_value`	numeric text	Maximum value reached before first 1.
`reached_one`	boolean	Whether the computation reached 1.
`termination_reason`	text	Reason computation stopped, usually reached one.
`computed_at`	timestamp	Time at which the result was computed.
`stored_at`	timestamp	Time at which the result was persisted.
`batch_id`	text or UUID	Optional identifier for the batch that produced the result.
`worker_id`	text	Optional identifier for the worker instance.

The result table should store completed values only when the computation has reached a valid termination condition. If a worker is interrupted before the result is complete, that incomplete attempt should be logged separately rather than inserted as a completed result.

For public reporting, `total_stopping_time`, `trajectory_length`, and `peak_value` are the most important per-integer fields. These values allow independent recomputation to compare the stored

output against a separate implementation.

6.5 Record Events Table

The `record_events` table stores moments when a newly evaluated starting value exceeds a previously known aggregate record. This table preserves a historical sequence of record changes rather than only the latest record.

A reference schema is shown in Table 3.

Table 3: Reference fields for record events.

Field	Type	Purpose
`id`	UUID or integer	Unique record event identifier.
`event_type`	text	Type of record, such as longest trajectory or highest peak.
`start_value`	integer or numeric text	Starting value that produced the record.
`record_value`	integer or numeric text	New record value.
`previous_record_value`	integer or numeric text	Previous stored record value, if available.
`result_id`	UUID or integer	Optional reference to the per-integer result.
`created_at`	timestamp	Time at which the record event was stored.

Record events are useful for dashboard feeds and historical analysis. They also help avoid losing information when aggregate records are overwritten. The `engine_state` table may store the current highest values, but `record_events` stores the path by which those records were reached.

Record events should be interpreted within the checked finite range. A new record event means that the value exceeds previous records observed by the engine. It does not mean that the value is a global record over all positive integers unless the checked range and prior literature support that claim.

6.6 Activity Logs Table

The `engine_activity_logs` table stores operational events. These events are not mathematical results. They describe system behavior such as start, stop, pause, resume, error, retry, lock acquisition, lock expiration, and batch completion.

A reference schema is shown in Table 4.

Activity logs support auditability and debugging. They should not be used as a substitute for result records. A log entry saying that a batch completed is useful, but the actual per-integer results or

Table 4: Reference fields for engine activity logs.

Field	Type	Purpose
`id`	UUID or integer	Unique log record identifier.
`event_type`	text	Operational event type.
`message`	text	Human-readable description of the event.
`severity`	text	Informational, warning, error, or critical.
`batch_id`	text or UUID	Optional batch identifier.
`worker_id`	text	Optional worker identifier.
`metadata`	JSON	Optional structured diagnostic data.
`created_at`	timestamp	Time at which the event was recorded.

aggregate state update should still be persisted in the relevant tables.

6.7 Run Intervals and Runtime Accounting

If the system needs accurate active runtime, it should store run intervals. A run interval records a period during which the engine was active or expected to be active. A reference interval contains:

- a start timestamp;
- an end timestamp, if the interval has ended;
- a status or end reason;
- optional worker or deployment metadata.

The total active runtime can then be computed as the sum of completed intervals plus the current open interval, if the engine is running. This is more reliable than using a browser-side timer.

For a set of completed intervals $[s_i, e_i]$, active runtime may be computed as

$$R = \sum_i (e_i - s_i).$$

If there is an active interval with start time s_{active}, and the engine is currently running at time t_{now}, then the displayed active runtime may be computed as

$$R_{\text{display}} = R + (t_{\text{now}} - s_{\text{active}}).$$

This computation should be performed using server time or database time where possible. Client-side rendering may display the value, but the source timestamps should be persistent.

6.8 Relationships Between Tables

The tables should be related in a way that supports inspection. A record event should be traceable to the result that produced it. A batch completion log should be traceable to the batch identifier used in result records. Aggregate state should be consistent with the stored results.

A simplified relationship model is shown in Figure 2.

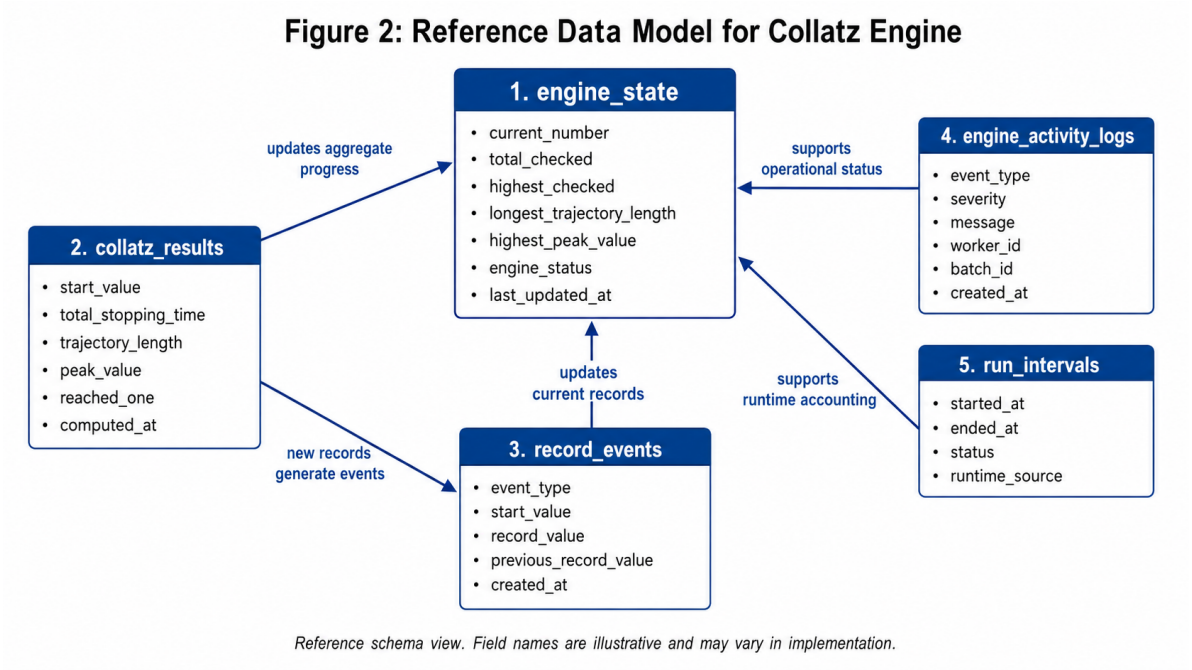

Figure 2: Reference data model for Collatz Engine. The diagram shows the main stored entities used for persistent engine state, per-integer computation results, record events, activity logs, and runtime intervals.

The diagram is intentionally simple. The goal is not to prescribe a final database schema. The goal is to separate computation state, result storage, record history, and operational diagnostics.

6.9 Constraints and Indexes

Database constraints help preserve the integrity of the computation record. At minimum, the results table should prevent duplicate completed results for the same starting value within the same computation stream. This can be implemented with a unique constraint on `start_value`, or on `engine_id` and `start_value` if multiple engines are supported.

Useful constraints and indexes include:

- a unique constraint on completed result starting values;
- an index on `start_value` for range inspection;
- an index on `computed_at` or `stored_at` for chronological queries;

- an index on `event_type` and `created_at` for record-event feeds;
- an index on `engine_status` if multiple engine records exist;
- constraints preventing negative starting values, stopping times, or trajectory lengths.

The database should also preserve the convention that starting values are positive integers. Invalid inputs should be rejected before computation or marked as operational errors, not stored as mathematical results.

6.10 Storage of Large Integers

Collatz trajectories can reach values much larger than their starting values. As the checked range increases, peak values may exceed ordinary integer limits in some programming or database environments.

The storage layer should preserve exact integer values. Approximate floating-point storage is not appropriate for starting values, stopping times, or peak values. A rounded peak value cannot support exact recomputation or audit.

There are two common approaches.

The first approach is to use a database numeric type with sufficient precision. This can preserve exact integer values if configured and handled correctly.

The second approach is to store large integers as text and validate that the stored text represents a nonnegative integer. This approach avoids precision loss during serialization, especially when the application layer uses arbitrary-precision integer values such as `BigInt`.

Whichever approach is used, the application should avoid converting large integer values into unsafe floating-point numbers during computation, serialization, or display.

6.11 Retention and Aggregation

The system may not need to store every full trajectory indefinitely. Full trajectories can be large, and storing them for every starting value may create unnecessary storage cost. For many reporting and verification purposes, the starting value, total stopping time, trajectory length, peak value, and termination status are sufficient.

However, the system should preserve enough information to support independent recomputation. If a reader can recompute a starting value and compare the stored stopping time and peak value, then the stored result has practical audit value.

A tiered storage approach may be appropriate:

- store summary results for every checked starting value;

- store full trajectories only for record events or sampled values;
- store aggregate time-series data for dashboard charts;
- archive periodic snapshots for public release.

This approach balances storage cost with transparency. It also allows the system to preserve important records without treating every trajectory as equally valuable for long-term storage.

6.12 Data Access and Public Reporting

The public dashboard should read from controlled views or API routes rather than directly exposing internal tables. This allows the system to present stable public fields while preserving internal implementation flexibility.

Public API responses may include:

- current number;
- total checked;
- highest checked;
- longest trajectory record;
- highest peak record;
- engine status;
- last updated time;
- recent record events;
- throughput estimates.

Internal fields such as worker locks, retry counts, diagnostic messages, and raw metadata should be exposed only where appropriate. Public reporting should provide enough information to interpret the computation without exposing details that could interfere with operation.

6.13 Summary

The data model separates persistent engine state, per-integer results, record events, and operational logs. This separation supports resume behavior, public reporting, auditability, and future verification work.

The central design rule is that durable computation should be represented in storage before it is reported as completed. The dashboard may display derived values, but those values should be traceable to persistent state and stored results.

7 Integrity, Reproducibility, and Auditability

7.1 Purpose of Integrity Controls

Integrity controls are required because Collatz Engine is a public computational system. The system reports numerical progress, record events, and finite verification status. If those values are not produced and stored under clear rules, the public interface can appear precise while the underlying computation remains difficult to inspect.

In this report, integrity means that the system preserves the correctness and continuity of its computation record. Reproducibility means that an independently implemented computation can evaluate the same starting value or range and obtain matching results under the same definitions. Auditability means that stored results, state transitions, and public dashboard values can be traced to durable data.

These requirements are operational. They do not turn a finite computation into a proof of the Collatz conjecture. They only support confidence that the finite computation reported by the engine corresponds to the values that were actually evaluated and stored.

7.2 Deterministic Computation

The Collatz map is deterministic. For a fixed starting value n, and for a fixed definition of the map, the trajectory is uniquely determined. This property makes the computation suitable for independent recomputation.

A deterministic result for n should include enough information to compare against a separate implementation. At minimum, this includes:

- the starting value n;
- the total stopping time;
- the recorded trajectory length;
- the peak value;
- the termination status.

If another implementation applies the same map and uses the same counting conventions, these values should match. If they do not match, the discrepancy should be investigated before the value is used in a public claim.

Determinism also requires exact arithmetic. If an implementation uses a numeric type that silently overflows or loses integer precision, then the computed trajectory may no longer correspond to the

Collatz map. The system should therefore use integer representations that preserve exact values over the checked range and over all intermediate values encountered in the trajectory.

7.3 Definition Consistency

Several quantities associated with Collatz trajectories have naming conventions that can vary across implementations. One implementation may report the number of map applications required to reach 1. Another may report the number of values listed in the trajectory. These differ by 1 when the trajectory reaches 1.

For this reason, Collatz Engine should preserve explicit definitions for the quantities it reports. In this report:

- total stopping time counts map applications;
- trajectory length counts recorded values from the starting value through the first occurrence of 1;
- peak value includes the starting value and all intermediate values before the first occurrence of 1;
- reaching 1 is the operational termination condition.

The dashboard and data exports should use the same conventions. If a public label says "longest trajectory," the report should make clear whether this refers to trajectory length or total stopping time. This avoids off-by-one ambiguity in later verification.

7.4 Sequential Coverage Assumption

The initial engine uses sequential evaluation. Under this model, the public checked range is meaningful only if every starting value in the range has been evaluated and persisted.

Suppose the engine reports that it has checked all values through B. The implied bounded statement is:

$$\forall n \in \mathbb{Z}_{>0},\ n \leq B \Rightarrow \exists k \geq 0 \text{ such that } T^k(n) = 1.$$

This statement depends on coverage. It is not enough for the engine to have evaluated many values below B. It must have evaluated every value in the interval.

The system should therefore distinguish between:

- total results stored;
- highest starting value checked;

- highest contiguous checked range;
- incomplete or retried batches.

In a strictly sequential system with no gaps, these values may be closely related. In a distributed or partially recovered system, they may differ. The public dashboard should avoid claiming contiguous coverage unless the stored state supports that claim.

7.5 Duplicate Prevention

Duplicate computation is not necessarily harmful if duplicates are detected and reconciled. It becomes harmful when duplicate results cause counters, record events, or public totals to be inflated.

Duplicate prevention should be enforced at more than one level.

At the database level, the results table should enforce uniqueness for completed results. In a single-engine system, a unique constraint on `start_value` may be sufficient. In a multi-engine system, the unique key may need to include `engine_id` and `start_value`.

At the worker level, range selection should prevent two active workers from claiming the same range. This may require a lock, lease, or atomic range assignment.

At the update level, aggregate counters should count logical completed starting values, not raw insert attempts. If a batch is retried and the same values are written again through idempotent logic, the counter should not increase incorrectly.

A duplicate-safe system should satisfy the following integrity condition: the reported value of `total_checked` should equal the number of distinct starting values with one completed result in the checked range.

This condition is operational. It states that the reported count should correspond to stored completed results, not to worker attempts or duplicate writes.

7.6 Gap Detection

A gap occurs when the engine reports progress beyond a starting value that does not have a completed result. In a sequential public computation, gaps weaken the interpretation of the checked range.

For a claimed checked range $1 \leq n \leq B$, every starting value from 1 through B should have a completed stored result. If any value in that interval is missing, then the public system should not describe the range as fully checked.

In practice, checking every value repeatedly may be expensive as the dataset grows. The system may therefore use a combination of methods:

- unique constraints on starting values;
- batch range records;
- periodic range audits;
- aggregate counts compared against expected range size;
- reconciliation queries for missing starting values;
- independent recomputation of sampled ranges.

The simplest strong condition is that the number of stored completed results from 1 through B equals B, and that the minimum and maximum starting values match the expected range. More detailed checks may be required if the system uses distributed workers or nonsequential ranges.

7.7 Crash Recovery and Durable Progress

Crash recovery is part of computation integrity. A worker may stop after selecting a range but before writing results. It may write some results but fail before updating aggregate state. It may update a heartbeat but fail to advance progress. Each case requires a recovery rule.

The main principle is that public progress should be advanced only when the corresponding computation has been persisted. A computed value in process memory is not durable progress.

For a batch beginning at c with length b, the system should avoid advancing the stored next value to $c + b$ until the completed results for the batch are durable or recoverable. If a crash occurs before the write, the engine should be able to retry the same range. If a crash occurs after partial writes, the engine should use idempotent writes or reconciliation logic to complete the range without double-counting.

Crash recovery should prefer repeated computation over skipped computation. Recomputing a deterministic value is usually acceptable. Skipping a value and reporting a contiguous checked range is not acceptable.

7.8 Record Event Integrity

Record events should be derived from completed results. A longest-trajectory record should correspond to a stored result whose trajectory length exceeds the previous stored maximum. A highest-peak record should correspond to a stored result whose peak value exceeds the previous stored maximum.

A record event should include enough information to be inspected later:

- event type;
- starting value;

- new record value;
- previous record value, if available;
- timestamp;
- reference to the result record, if available.

The aggregate record stored in engine state should be consistent with the latest relevant record event. If the engine state reports a highest peak value, there should be a result or event that supports it.

Record event integrity is important because record feeds are likely to attract public attention. They should not be generated from incomplete computation, duplicate inserts, or temporary values that were never persisted.

7.9 Dashboard Consistency

The public dashboard should display values derived from persistent state and stored results. A dashboard value may be cached, formatted, or animated, but it should remain traceable to durable data.

Dashboard consistency includes several requirements.

First, counters should not advance beyond persisted progress. If the dashboard increments a visible total in the browser before the database update completes, the display may temporarily exceed the durable computation record.

Second, runtime values should be based on stored timestamps and run intervals. A browser-side timer alone cannot represent engine uptime across deployments or downtime.

Third, record values should be read from aggregate state or record-event tables, not inferred from a local visualization.

Fourth, status should reflect stored operational evidence. If the engine has not produced a heartbeat or progress update within the expected interval, the dashboard should not continue to present the system as normally running without qualification.

The dashboard is part of the audit surface. It should make finite computation visible without overstating what the computation establishes.

7.10 Independent Recalculation

Because the Collatz map is deterministic, independent recalculation is a practical audit method. An external verifier can select a starting value or range, compute the trajectory under the same definitions, and compare the result with stored values.

For a single starting value n, an independent verifier should be able to compare:

- $\sigma_\infty(n)$, the total stopping time;
- $L(n)$, the recorded trajectory length;
- $P(n)$, the peak value;
- the termination status.

For a range $1 \leq n \leq B$, an independent verifier may compare aggregate values such as the longest trajectory, highest peak, and number of completed results. The verifier may also recompute the entire range if the range is small enough, or sample subranges if the range is large.

Independent recalculation does not prove the conjecture. It verifies that the stored finite computation is consistent with a separate implementation over the checked values.

7.11 Audit Logs

Audit logs should record operational events that affect interpretation of the computation. These include engine start, pause, resume, worker lock acquisition, lock release, batch completion, retry, error, and recovery events.

An audit log should not replace result storage. It should explain how the system operated around the results. For example, a batch completion log is useful, but it should be supported by the actual stored results or the aggregate state update associated with the batch.

Useful log fields include:

- event type;
- severity;
- timestamp;
- worker identifier;
- batch identifier;
- affected starting range;
- message;
- structured metadata.

Audit logs are especially useful when the engine recovers from interruption. They provide a record of what the system believed happened and what recovery action was taken.

7.12 Reconciliation Procedures

Reconciliation is the process of comparing stored state, result records, and logs to identify inconsistencies. It should be possible to run reconciliation periodically or after an interruption.

A reconciliation procedure may check:

- whether `total_checked` matches the count of completed result records;
- whether `highest_checked` matches the highest contiguous completed starting value;
- whether aggregate record values match the maximum values in the results table;
- whether record-event history is consistent with aggregate records;
- whether any claimed batch ranges are incomplete;
- whether duplicate results exist;
- whether recent status matches heartbeat and progress timestamps.

If a discrepancy is found, the system should prefer a conservative public state. For example, if the aggregate count is higher than the number of completed results, the system should not report the higher value as verified progress without explanation.

7.13 Integrity Boundaries

The integrity controls described in this section apply to the system's finite computation record. They do not establish the truth of the Collatz conjecture. They only help ensure that when the system reports a checked range, record event, or computed value, that report is supported by durable data.

There are also limits to what the current system can establish. If the implementation is closed, external readers may be able to inspect public outputs but not the full computation path. If full result exports are not available, independent verification may require selected recomputation rather than complete audit. If the system uses a single worker, it avoids some distributed consistency problems but may have throughput limits.

These boundaries should be stated openly. A system can be useful as a public computational observatory while still having defined operational limits.

7.14 Summary

Integrity, reproducibility, and auditability are design requirements for Collatz Engine. The system should compute deterministically, store durable results, prevent duplicate counting, detect gaps, recover conservatively after interruption, and expose public values that are traceable to persistent data.

These controls support finite verification over checked ranges. They do not convert finite computation into proof. The distinction remains central to the interpretation of the engine.

8 Public Dashboard and Visualization Layer

8.1 Purpose of the Public Dashboard

The public dashboard is the reporting layer of Collatz Engine. It presents selected values from the persistent computation state and makes the running system visible to external readers. The dashboard is not the authority for computation. It reads from stored state, formats the values, and displays them.

The dashboard has three main purposes.

First, it reports the current operational state of the engine. A reader should be able to determine whether the engine is running, paused, stalled, or unavailable.

Second, it reports finite computational progress. A reader should be able to see the current starting value, total checked values, checked range, and current aggregate records.

Third, it provides visual summaries of trajectory behavior. These may include a current trajectory view, record-event feed, heatmaps, descent events, and historical charts.

The dashboard should not imply that the Collatz conjecture has been proved. Its values describe finite computation performed by the engine.

8.2 Dashboard as a Reporting Surface

The dashboard should be treated as a reporting surface, not as a computation source. The browser may animate values or update visible counters, but the authoritative values should come from the database or from server-side API responses derived from the database.

This distinction is important for long-running computation. If a page is refreshed, closed, or reopened, the dashboard should recover its displayed state from persistent data. If the worker stops, the dashboard should not continue presenting progress as if computation were still advancing. If the engine resumes after downtime, the dashboard should display resumed state based on stored timestamps and counters.

A public dashboard for Collatz Engine should therefore follow a simple rule:

Displayed progress should be traceable to persistent state.

This rule does not prevent client-side rendering. It only means that visible progress should not be invented by the browser.

8.3 Core Dashboard Fields

The dashboard should include a small set of core fields that describe the state of the computation. These fields should be named clearly and should correspond to values stored in persistent state or derived from stored results.

A reference set of dashboard fields is shown in Table 5.

Table 5: Reference fields for the public dashboard.

Dashboard field	**Meaning**
Engine status	Current operational status, such as running, paused, stalled, or error.
Current number	The next or current starting integer under evaluation, depending on implementation convention.
Total checked	Number of completed and persisted starting values.
Highest checked	Highest starting value included in the completed checked range.
Runtime	Active runtime derived from stored timestamps or run intervals.
Throughput	Estimated number of completed starting values per unit time.
Longest trajectory	Largest recorded trajectory length observed in the checked range.
Longest trajectory start	Starting value that produced the current longest trajectory record.
Highest peak	Largest intermediate value observed in the checked range.
Highest peak start	Starting value that produced the current highest peak record.
Last updated	Most recent timestamp at which durable progress was written.

The field labels should avoid ambiguity. If the dashboard displays "current number," the documentation should state whether this is the next starting integer to be evaluated or the starting integer currently being processed. If the dashboard displays "longest trajectory," it should state whether the value is trajectory length or total stopping time.

8.4 Persistent Runtime Clock

The runtime clock should be based on persistent timestamps. A browser-only timer is not sufficient because it measures the duration of the page session rather than the duration of the engine's operation.

A persistent runtime clock should use stored run intervals or stored start and pause timestamps. If the engine is running, the displayed runtime may be updated in the browser, but the calculation

should be anchored to database timestamps. If the engine is paused, stalled, or offline, the clock should reflect that state rather than continuing without qualification.

The dashboard should distinguish between active runtime and wall-clock age.

Active runtime refers to time during which the engine was running or expected to be running.

Wall-clock age refers to the elapsed time since the system or state record was created.

These are not the same quantity. A public report should not describe wall-clock age as computation runtime unless the engine was actually running throughout that period.

8.5 Current Number and Checked Range

The current number is one of the most visible fields on the dashboard. It should be defined carefully.

If the system stores the next starting value to be evaluated, then the field should be described as the next starting value. If the system stores the value currently being processed by a worker, then the field should be described as the current starting value. These two conventions differ during execution.

For public interpretation, the checked range is usually more important than the transient current value. If the engine has completed all starting values through B, then the dashboard can report that the checked range extends through B, subject to the integrity controls described earlier.

The dashboard should avoid claiming a checked range based only on a current counter. A checked range requires completed and persisted results for each starting value in the interval.

8.6 Total Checked

The total checked field should count completed and persisted starting values. It should not count values merely selected for computation. It should not count duplicate attempts. It should not count values that were computed in memory but not stored.

In a sequential system without gaps, total checked may correspond to the highest checked starting value if the system begins at 1. In other models, total checked and highest checked may differ. For example, a distributed system may have completed many values while leaving gaps in the range. The dashboard should report these quantities separately if the computation model requires it.

For the initial sequential engine, the safest public statement is:

Total checked means completed and persisted starting values.

This wording ties the displayed count to durable computation rather than transient execution.

8.7 Longest Trajectory Display

The longest trajectory display reports the largest trajectory length observed within the checked range. It should include both the record value and the starting value that produced it.

For example, the dashboard may display:

- longest trajectory length;
- starting value that produced the longest trajectory;
- timestamp of the record event;
- previous record value, if shown in a record feed.

This display should not be described as a global mathematical record unless the system has verified that claim against the relevant external record history. Within the context of Collatz Engine, the phrase should mean the longest trajectory observed by the engine in its checked range.

The dashboard should also preserve the distinction between trajectory length and total stopping time. If the system reports trajectory length, the value includes the starting value and the first occurrence of 1. If it reports total stopping time, the value counts applications of the Collatz map.

8.8 Highest Peak Display

The highest peak display reports the largest intermediate value observed within the checked range. It should include the peak value and the starting value that produced it.

Peak values can become large. The dashboard should format them in a way that preserves exactness. Abbreviated display may be useful for readability, but the exact value should remain available through a detail view, tooltip, expandable field, or data export.

For example, the dashboard may show a shortened value in the main interface but allow the reader to inspect the full integer. The shortened value should not replace the stored exact value.

The highest peak display should also make clear that the value is finite and observational. It is the highest peak encountered by the engine so far, not a statement about all possible trajectories.

8.9 Throughput Reporting

Throughput measures how quickly the engine completes starting values. It may be reported as values checked per second, per minute, or per hour.

Throughput should be computed over a defined interval. A lifetime average, a recent rolling average, and an instantaneous estimate may produce different values. The dashboard should avoid presenting throughput without indicating how it is calculated.

A simple recent throughput estimate can be computed from two stored observations. If the engine had checked C_1 values at time t_1, and C_2 values at time t_2, then the average throughput over that interval is

$$\frac{C_2 - C_1}{t_2 - t_1}.$$

This value should be interpreted operationally. It depends on worker speed, database writes, batch size, deployment environment, and current load. It is not a mathematical property of the Collatz map.

8.10 Trajectory Visualization

The trajectory visualization displays the behavior of a selected starting value under repeated application of the Collatz map. This may be the current value being evaluated, a recent record value, or a user-selected starting value.

A trajectory visualization may show:

- the sequence of values;
- the rise and descent of the trajectory;
- the peak value;
- the step at which the peak occurs;
- the descent into $16, 8, 4, 2, 1$;
- the total stopping time or trajectory length.

The visualization should be treated as a derived view. It helps readers inspect behavior, but it does not replace stored results. If a visualization is generated on demand, the system should state whether it is based on stored trajectory data or recomputed from the starting value.

8.11 Heatmaps and Aggregate Visualizations

Aggregate visualizations can show patterns in finite computed data. Examples include heatmaps of stopping times, distributions of peak ratios, record-event timelines, and descent-event summaries.

These visualizations may help readers inspect the computed range, but they should not be presented as proof of a general pattern. A heatmap over a finite range describes that range. It does not establish the behavior of all positive integers.

Useful aggregate visualizations include:

- stopping time distribution over the checked range;

- peak value distribution over the checked range;
- record-event timeline;
- recent throughput chart;
- trajectory length by starting value over a selected interval;
- sampled parity patterns;
- descent moments where a trajectory first falls below its starting value.

Each visualization should identify its data scope. A chart should state the range, sample size, or time interval it represents.

8.12 Record Event Feed

A record event feed lists moments when the engine observes a new longest trajectory or highest peak. This feed is useful because record events are sparse compared with ordinary per-integer results.

Each feed item should include:

- event type;
- starting value;
- new record value;
- previous record value, if available;
- timestamp;
- link or reference to the associated result, if available.

The feed should avoid exaggerated language. A record event means that the engine has observed a new record within its own checked range. It should not be described as a new mathematical discovery unless the claim is independently supported.

8.13 Status, Staleness, and Error Display

The dashboard should make operational status visible. If the engine is running normally, the dashboard may display that state. If progress has not updated recently, the dashboard should indicate that the state may be stale. If the worker has failed, the dashboard should show an error or degraded status.

Status should be based on persistent evidence such as:

- stored engine status;

- last update timestamp;
- last heartbeat timestamp;
- recent activity logs;
- worker lock state;
- recent error records.

The dashboard should not silently hide operational failure. A public computation becomes more credible when downtime and recovery are visible rather than disguised.

8.14 Figure Placeholder for Dashboard

A screenshot or schematic of the dashboard should be included in the final report. The figure should show the actual public interface or a simplified annotated version of it. It should not be a decorative illustration.

Figure placeholder: Public dashboard screenshot

Insert a screenshot showing runtime, current number, total checked values, longest trajectory, highest peak, throughput, and record-event feed.

The final figure should be captured from the live Collatz Engine dashboard.

Figure 3: Public dashboard reporting finite computational progress, operational status, and observed record values.

This figure should be updated near publication so that it reflects the deployed system at the time of release.

8.15 Interpretation Constraints

The dashboard should be designed so that a reader can distinguish computation, visualization, and mathematical claim.

The following language is appropriate:

- “checked values”;
- “observed record”;
- “finite computation”;
- “trajectory reached 1”;
- “current checked range”;
- “highest peak observed by the engine.”

The following language should be avoided:

- “Collatz solved”;
- “proof in progress”;
- “closing in on the conjecture”;
- “guaranteed convergence for all integers”;
- “mathematical breakthrough.”

This language constraint is part of the system design. Public presentation should not assign more meaning to the computation than the computation supports.

8.16 Summary

The public dashboard reports the state of Collatz Engine. It should display persistent computation state, finite progress, observed records, runtime, throughput, and selected visualizations. Its values should be traceable to durable storage.

The visualization layer helps readers inspect trajectory behavior and aggregate patterns over checked ranges. It does not prove the Collatz conjecture. The dashboard should preserve that distinction in its labels, charts, captions, and public summaries.

9 Current Computational Status

9.1 Purpose of This Status Section

This section records the public computational status shown by the Collatz Engine dashboard at the time of the dashboard screenshot used for this report. The screenshot is treated as a public reporting snapshot. It is not treated as a direct database export.

The purpose of this section is to document what the public system displayed at the time of capture. The values reported here describe finite computation shown by the engine dashboard. They do not constitute a proof of the Collatz conjecture.

The screenshot used for this section was captured from the Collatz Engine public interface on June 11, 2026 at approximately 10:13 AM Pacific time, based on the visible local time and dashboard status fields.

9.2 Dashboard Snapshot

Figure 4 shows the public Collatz Engine dashboard used for this status section. The screenshot includes the current computation display, public progress counters, trajectory visualizations, record tables, aggregate charts, and supporting dashboard sections.

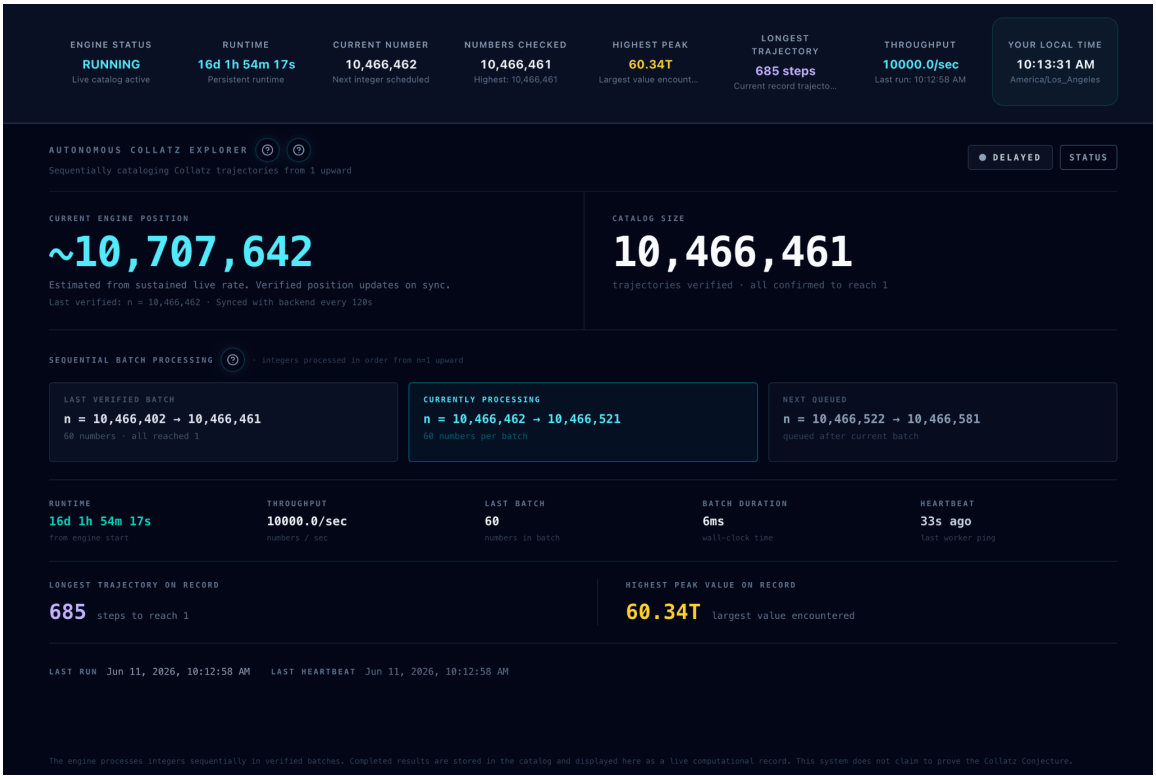

Figure 4: Public status snapshot from Collatz Engine showing current engine status, runtime, checked values, current processing range, throughput, heartbeat, and aggregate record values at the time of capture.

The screenshot is useful as a publication artifact because it records the public state of the engine at a specific time. It should not be treated as a substitute for a database export. Small dashboard fields that are not legible in the screenshot are marked as not confirmed rather than reconstructed manually.

9.3 Reported Public Dashboard Values

The most legible public values visible in the dashboard screenshot are the large progress counters near the top of the engine interface. These values are recorded in Table 6.

Table 6: Public dashboard status values visible in the Collatz Engine status snapshot.

Field	**Reported value from status snapshot**
Screenshot capture context	June 11, 2026, approximately 10:13 AM Pacific time
Engine status	Running
Runtime	16d 1h 54m 17s
Current number	10,466,462
Numbers checked	10,466,461
Catalog size	10,466,461
Current engine position	approximately 10,707,642
Highest peak value displayed	60.34T
Longest trajectory displayed	685 steps
Throughput displayed	10000.0/sec
Last verified batch	$n = 10,466,402 \to 10,466,461$
Currently processing	$n = 10,466,462 \to 10,466,521$
Next queued	$n = 10,466,522 \to 10,466,581$
Last batch size	60
Batch duration	6ms
Heartbeat age	33s ago
Last run	Jun 11, 2026, 10:12:58 AM
Last heartbeat	Jun 11, 2026, 10:12:58 AM

The values in Table 6 should be interpreted as values visible on the public dashboard screenshot. They should not be described as database-audited values unless they are later checked against the production database or an API export.

9.4 Checked Range Interpretation

The dashboard screenshot shows the engine operating in a range above ten million checked or displayed values. The large public counters visible in the screenshot indicate a current number of 10,466,462 and a checked catalog size of 10,466,461.

This report does not infer a complete checked interval solely from the screenshot. In a sequential computation, the checked range should be derived from persistent state and completed result records. A screenshot can show what the dashboard displayed, but it cannot by itself prove that every starting value in an interval has a completed stored result.

For that reason, the current report uses the following conservative interpretation:

> At the time of the dashboard screenshot, Collatz Engine publicly displayed computation progress above ten million evaluated or reported values. The screenshot records public dashboard state, but contiguous range verification should be confirmed from persistent engine data before making a bounded verification claim.

This wording preserves the distinction between public display and database-backed verification.

9.5 Trajectory and Visualization Status

The screenshot shows that the public dashboard includes trajectory visualization. The visible trajectory chart displays a computed path with a peak marker and a descending sequence over subsequent steps. This supports the description of Collatz Engine as a system that does more than count checked values. It also visualizes trajectory behavior for selected values.

The screenshot also shows additional aggregate charts lower on the page. These include chart sections that appear to represent historical or distributional views of computed data. These visualizations are useful for public inspection, but they should be interpreted as views over finite data.

The screenshot does not provide enough legible detail to extract exact chart values for this report. The presence of these visualizations is recorded, but their numerical contents are not reconstructed manually.

9.6 Result Tables and Record Sections

The screenshot includes table-based sections that appear to list computed values, records, or recent engine outputs. These sections are relevant to the data model described earlier in the report because they show how stored computation can be surfaced publicly.

However, the individual rows in the screenshot are not legible enough to extract exact record-event values. This report therefore does not include a manually reconstructed recent record-event table. A future version should generate such a table directly from stored record-event data.

The correct publication approach is:

- use screenshots to document the public interface;
- use database or API exports to populate exact numerical tables;

- avoid reconstructing small table values manually from compressed screenshots.

This avoids introducing transcription errors into the report.

9.7 Operational Status

The dashboard screenshot shows the public engine interface in an active rendered state. The page includes visible progress counters, dashboard cards, charts, tables, and record sections. This indicates that the public reporting layer was available at the time of capture.

This report does not treat the screenshot as proof that the worker was actively computing at the exact moment of capture. Worker activity should be confirmed from persistent status fields, heartbeat timestamps, or recent durable updates. The screenshot confirms public visibility. It does not, by itself, confirm worker heartbeat state.

For publication, the operational statement should therefore remain conservative:

> At the time of the screenshot, the public dashboard was available and displayed active computation status and progress counters. Worker activity and last durable update should be confirmed from persistent engine state before publication.

9.8 Interpretation of Current Results

The dashboard values visible in the screenshot document finite computational behavior reported by Collatz Engine. They do not prove the Collatz conjecture. They also do not replace persistent state, result records, or independent verification.

The correct interpretation is that the screenshot records a public status snapshot of the engine. It shows that the system was publicly presenting computation progress, trajectory visualization, aggregate charts, and result sections at the time of capture.

The report should not state that the conjecture has been solved. It should not imply that the displayed counters establish an unbounded mathematical claim. The displayed values are finite computational status values.

9.9 Data Quality Notes

The screenshot is suitable for documenting public dashboard state, but it has limits as a data source.

First, several small numerical fields are not legible enough to transcribe safely.

Second, the screenshot records public display state, not direct database state.

Third, exact record-event values should be generated from stored data rather than manually read from the image.

Fourth, last durable update, heartbeat status, and throughput method should be confirmed from the production database or API before final publication.

For a publication-ready version of this chapter, the following data should be exported directly from the engine:

- current number;
- total checked;
- highest checked value;
- longest trajectory length;
- starting value for longest trajectory;
- highest peak value;
- starting value for highest peak;
- last durable update timestamp;
- recent throughput interval and value;
- recent record events.

Until those values are exported, this chapter should be described as a screenshot-derived public status snapshot.

9.10 Summary

The public Collatz Engine screenshot shows a working dashboard with visible computation progress, trajectory visualization, aggregate charts, record sections, and supporting public interface elements. The most legible large values shown are a current number of 10,466,462, a checked catalog size of 10,466,461, a displayed highest peak of 60.34T, a longest trajectory record of 685 steps, and a displayed throughput of 10000.0 values per second.

This chapter records those visible values conservatively. It does not infer unconfirmed values from small text in the screenshot, and it does not treat the screenshot as a database audit. The screenshot documents public status at the time of capture. Exact publication values should be confirmed from persistent engine data before final release.

10 Limitations

10.1 Purpose of This Section

This section states the limitations of Collatz Engine as a computational system. These limitations are not defects in the project. They define the boundary between finite computation, public reporting, and mathematical proof.

The Collatz conjecture is a universal statement about all positive integers. Collatz Engine evaluates finite starting values under the Collatz map and records the observed results. No finite checked range, by itself, proves the full conjecture.

For that reason, this report treats limitations as part of the system design. The dashboard, documentation, and published summaries should preserve the distinction between what the engine has computed and what remains mathematically unresolved.

10.2 Finite Computation Is Not Proof

The main limitation is mathematical. Collatz Engine can verify that particular starting values reach 1. It can also verify that every starting value in a finite checked range reaches 1, assuming the computation and stored records are correct. It cannot prove the conjecture for all positive integers by finite enumeration alone.

If the engine has checked all starting values through B, the strongest bounded statement is:

$$\text{For every positive integer } n \leq B, \text{ the computed trajectory of } n \text{ reaches 1.}$$

The full conjecture requires the unbounded statement:

$$\text{For every positive integer } n, \text{ the trajectory of } n \text{ reaches 1.}$$

These statements are not equivalent. Increasing B strengthens the finite verification range, but it does not remove the need for a proof or a counterexample in the general case.

The public system should therefore avoid language that implies mathematical closure. It should not present computation as proof. It should present computation as finite verification over specified ranges.

10.3 Implementation Correctness

The reliability of the engine depends on the correctness of its implementation. The Collatz map is simple to define, but implementation errors can still occur. Examples include off-by-one errors in

stopping time, incorrect parity checks, incorrect handling of large integers, failed database writes, duplicate counting, and inconsistent state updates.

The system's reported values are only as reliable as the computation and persistence path that produced them. A correct mathematical definition does not guarantee a correct implementation.

This limitation can be reduced through testing, independent recomputation, database constraints, reconciliation procedures, and public documentation of counting conventions. It cannot be ignored. If a public number is displayed, the system should be able to explain how that number was computed and where it is stored.

10.4 Numeric Precision and Overflow

Collatz trajectories can reach intermediate values much larger than their starting values. If the implementation uses a numeric type that cannot represent those values exactly, the computed trajectory may become incorrect.

This is especially relevant in JavaScript and TypeScript environments. The standard `number` type uses floating-point representation and cannot safely represent all integers beyond its safe integer limit. For exact Collatz computation over larger values, the implementation should use arbitrary-precision integer arithmetic, such as `BigInt`, or another exact integer representation.

The storage layer has the same limitation. If a peak value is stored in a type that rounds or truncates the value, then the stored result cannot support exact verification. Large starting values, peak values, and record values should be stored in a format that preserves exact integer values.

10.5 Persistence and State Consistency

The engine depends on persistent state to resume computation and report progress. If persistent state becomes inconsistent with stored results, public reporting may be affected.

For example, the engine state may report a total checked count that does not match the number of completed result records. It may report a highest checked value even though some starting values below that value are missing. It may report a longest trajectory value that is not supported by a stored result.

These are system consistency problems. They do not have mathematical significance, but they affect the credibility of public reporting.

The system should therefore support reconciliation. Aggregate state should be compared periodically against stored results and record events. If a discrepancy is found, the public dashboard should prefer conservative reporting until the discrepancy is resolved.

10.6 Throughput Constraints

Throughput is limited by the worker implementation, arithmetic cost, database write rate, deployment environment, network latency, and batch size. Trajectories do not all require the same amount of computation. Some starting values terminate quickly. Others produce longer trajectories and larger intermediate values.

As a result, throughput may vary over time. A short-term throughput value should not be interpreted as a stable benchmark unless the measurement interval and method are defined.

The current system should report throughput as an operational measurement. It should not imply that throughput is constant or that future performance will match a recent average.

10.7 Single-Worker Architecture

The initial implementation is based on a simple forward-moving computation model. This is appropriate for clarity and auditability, but it limits throughput compared with a distributed system.

A single-worker architecture avoids some coordination problems. It reduces the risk of overlapping ranges, duplicate workers, and inconsistent distributed state. It also makes the checked range easier to interpret.

The tradeoff is that progress is limited by the capacity of one worker process and its persistence path. Scaling to multiple workers would require additional design work, including range assignment, worker leases, duplicate prevention, reconciliation, and possibly independent verification of assigned ranges.

Distributed execution is therefore a future extension, not an assumption of the current report.

10.8 Dashboard Interpretation

The dashboard can make computation visible, but it can also create interpretation risk. Large counters, live clocks, animated trajectories, and record feeds may make the system appear more mathematically conclusive than it is.

This limitation is a communication problem, not a computation problem. A reader may see a large checked count and infer that the conjecture is close to being resolved. That inference is not valid.

The dashboard should therefore label values carefully. It should use terms such as “checked values,” “observed record,” “finite computation,” and “current checked range.” It should avoid language that suggests proof, inevitability, or mathematical closure.

10.9 Visualization Limits

Visualizations can help readers inspect computed data, but they are summaries of finite information. A heatmap, trajectory chart, record timeline, or descent visualization may reveal structure within a checked range. It does not establish a theorem about all positive integers.

Visualizations also involve design choices. A chart may use sampling, scaling, smoothing, aggregation, or truncation. These choices can affect how patterns appear. For that reason, visualizations should state their scope and method where necessary.

A visualization should be treated as an aid to interpretation, not as evidence beyond the data it represents.

10.10 Data Retention Limits

The system may not store every full trajectory indefinitely. Full trajectories can be large, and storing every intermediate value for every starting integer may become expensive. The engine may store summary results for every checked value while storing full trajectories only for selected records, recent computations, or sampled cases.

This creates a distinction between summary verification and full trajectory inspection. If only summary values are stored, an external verifier can still recompute a starting value and compare total stopping time and peak value. However, the stored database may not contain every intermediate step.

The report should make clear which data is stored permanently, which data is derived on demand, and which data is displayed temporarily.

10.11 External Verification Limits

Independent verification is possible because the Collatz map is deterministic. However, the scope of verification depends on what data is available.

If the system publishes only dashboard summaries, external verification is limited. If it publishes result exports, external readers can audit stored values more directly. If it publishes code, schema, and datasets, verification becomes stronger.

The current report documents the system but does not, by itself, establish a complete third-party audit. A stronger verification process would require controlled exports, checksum records, documented versions, and independent recomputation over selected ranges.

10.12 Operational Downtime

The engine may experience downtime because of deployment changes, worker crashes, database limits, network interruptions, hosting constraints, or maintenance. Downtime affects throughput

and current status. It does not affect the validity of already persisted completed results, assuming those records remain intact.

The dashboard should distinguish between a paused or stalled system and an active computation. Runtime accounting should also distinguish active runtime from wall-clock age.

If the engine is offline, the public interface should report that condition rather than continuing to display a misleading active state.

10.13 Scope of This Report

This report describes the current Collatz Engine system. It does not claim to survey the full mathematical literature on the Collatz conjecture. It does not compare the engine against all prior computational verification efforts. It does not present a new proof, a counterexample, or a new theorem.

The report is a technical documentation artifact. Its scope is the architecture, computation method, persistence model, data design, integrity controls, dashboard reporting, and finite computational status of the system.

Future reports may extend this scope to include larger verified ranges, exported datasets, distributed computation, independent verifier tools, or more detailed statistical analysis.

10.14 Summary

The central limitation of Collatz Engine is that it performs finite computation. It can record and report observed behavior over checked ranges. It cannot prove the Collatz conjecture by finite enumeration.

Other limitations are operational: implementation correctness, numeric precision, state consistency, throughput, downtime, data retention, and external verification. These limitations should be documented rather than hidden. Clear boundaries make the system more credible and reduce the risk of overstating what the computation establishes.

11 Future Work

11.1 Purpose of Future Work

This section describes possible extensions to Collatz Engine. These extensions are technical and operational. They do not change the mathematical status of the Collatz conjecture. A larger computation, a faster worker, or a more detailed visualization layer can improve the system as a computational observatory, but none of these additions constitutes a proof.

The future work described here is organized around five goals:

- increasing computational throughput;
- improving reproducibility and independent verification;
- expanding public data access;
- improving visualization and analysis;
- strengthening archival and citation practices.

These goals should be implemented in a way that preserves the existing design constraints: persistent state, clear definitions, durable results, conservative public reporting, and separation between finite computation and proof.

11.2 Distributed Workers

The current engine is based on a sequential computation model. This model is simple to inspect, but it limits throughput. A future version may distribute work across multiple workers.

Distributed execution would require a range-assignment system. Instead of one worker advancing a single current value, the system would assign finite intervals to workers. Each worker would compute results for its assigned interval and write completed results back to the database.

A distributed model introduces several additional requirements:

- non-overlapping range assignment;
- worker leases or locks;
- retry behavior for incomplete ranges;
- duplicate prevention;
- gap detection;
- result reconciliation;

- aggregate record updates from multiple workers.

The main risk in distributed computation is not the Collatz calculation itself. The map is deterministic. The main risk is coordination. If two workers process the same range, the system may double-count results. If a worker fails after receiving a range, the system may leave a gap. If aggregate records are updated without proper ordering, the dashboard may display inconsistent values.

For these reasons, distributed execution should be introduced only after the single-worker persistence model is stable. The system should first prove operationally that it can resume, reconcile, and report a sequential computation correctly.

11.3 Range Assignment and Verification

A future distributed version should represent work as explicit ranges. Each range should have a start value, end value, status, assignment timestamp, completion timestamp, worker identifier, and verification status.

A range may pass through states such as:

- unassigned;
- assigned;
- running;
- completed;
- failed;
- expired;
- verified.

A range should be marked completed only after all results in the range have been stored or after a compact verification record has been written. A range should be marked verified only after an additional check confirms that the range contains no missing results and that summary values match the stored computation.

This structure would allow the public dashboard to distinguish between total assigned work, completed work, and verified contiguous coverage. That distinction becomes important when computation is no longer strictly sequential.

11.4 Independent Verifier Tooling

A future release should include a separate verifier tool. The verifier should be able to recompute selected starting values or ranges and compare the computed results against stored engine results.

The verifier should use the same definitions stated in this report:

- total stopping time counts map applications;
- trajectory length includes the starting value and the first occurrence of 1;
- peak value includes the starting value and all intermediate values;
- reaching 1 is the termination condition.

The verifier should be independent from the primary worker code where practical. If the verifier shares all computation logic with the worker, it may repeat the same implementation error. A stronger verification path would use a separate implementation, possibly in a different language, to recompute selected ranges.

The verifier does not prove the conjecture. It improves confidence that the stored finite computation matches the defined Collatz map over checked values.

11.5 Public Data Exports

Public data exports would make the engine more inspectable. A future version should provide downloadable files containing completed result summaries over defined ranges.

A basic export row may include:

- starting value;
- total stopping time;
- trajectory length;
- peak value;
- completion timestamp;
- engine version or computation version.

Exports should be versioned. If the computation method, schema, or counting convention changes, the export should identify the version used. This prevents later readers from comparing values produced under different definitions without noticing the difference.

For larger exports, the system should include checksums. A checksum allows a reader to confirm that a downloaded file has not changed. If exports are updated periodically, each release should have a publication timestamp, range description, and checksum.

11.6 Public API Access

A public API may allow external readers, researchers, or developers to query engine status and selected result data. The API should expose stable fields and avoid exposing internal implementation details.

Useful API endpoints may include:

- current engine status;
- aggregate records;
- recent record events;
- result lookup by starting value;
- checked range summary;
- export metadata.

The API should preserve the same interpretation constraints as the dashboard. Field names should be explicit. A value named `trajectory_length` should not contain total stopping time. A value named `total_checked` should count completed and persisted results.

Rate limits may be required to protect the production system. Public access should support inspection without interfering with the running computation.

11.7 Improved Runtime Accounting

Future work should improve runtime accounting by recording run intervals explicitly. A run interval should include a start timestamp, end timestamp, status, and reason for ending. This would allow the dashboard to distinguish active runtime from wall-clock age.

More precise runtime accounting would support better throughput reporting. The system could report:

- lifetime average throughput;
- recent rolling throughput;
- batch-level throughput;
- active-runtime throughput;
- wall-clock throughput.

These values measure different things. A lifetime average over active runtime may differ from a wall-clock average that includes downtime. The dashboard and report should state which method

is being used.

11.8 Record Event Analysis

The record-event feed can be expanded into a more formal analysis layer. Future reports may study how often new longest-trajectory and highest-peak records occur within the checked range.

Possible analyses include:

- intervals between new trajectory-length records;
- intervals between new peak-value records;
- ratio of peak value to starting value;
- relation between stopping time and peak value;
- distribution of record-producing starting values;
- comparison between recent and historical record density.

These analyses should be presented as observations over finite data. They may reveal structure in the checked range, but they should not be stated as general results unless supported by proof.

11.9 Visualization Extensions

The visualization layer can be extended beyond the initial dashboard. Future visualizations may include more detailed trajectory plots, density maps, record-event timelines, descent-event views, and orbit-merging diagrams.

Useful additions include:

- trajectory comparison for selected starting values;
- peak-ratio charts;
- stopping-time histograms;
- record-event timeline;
- checked-range heatmap;
- descent-to-below-start visualization;
- convergence tree views for selected intervals.

Each visualization should state its data scope. If a chart uses sampled data, the sample method should be stated. If a chart uses the full checked range, the range should be stated. If a chart is generated from cached aggregate data, that should be clear where relevant.

Visualizations should remain secondary to the stored computation. They help readers inspect the data, but they are not the source of truth.

11.10 Code and Schema Publication

Public release of code and schema would improve auditability. If the system publishes the worker code, database schema, and computation definitions, external readers can inspect how results are produced.

A useful publication package may include:

- worker source code;
- database schema;
- migration files;
- result export format;
- verifier script;
- dashboard API field definitions;
- version history.

If full publication is not possible, the system can still publish selected artifacts. At minimum, the report should document the computation method and storage conventions clearly enough for an independent implementation to reproduce selected results.

11.11 Archival Releases and Citation

Future releases should be archived in a way that supports citation. A technical report, code release, or dataset export should have a stable version identifier. Where possible, archival releases should include a DOI.

An archival release may include:

- report PDF;
- source code snapshot;
- schema snapshot;
- dataset export;
- checksum file;
- citation metadata.

Stable release artifacts are useful because the live engine may continue to change. A reader citing the project should be able to refer to a specific report, code version, or dataset snapshot rather than an evolving webpage.

11.12 Future Reports

This report is intended as an initial technical record. Future reports may focus on narrower subjects.

Possible future reports include:

- a distributed worker architecture report;
- a data export and verification report;
- a record-event analysis report;
- a dashboard and visualization report;
- a public API specification;
- a report on independent verification methods.

Each future report should preserve the same central distinction. The engine can document finite computation. It should not state or imply that the conjecture has been proved unless a valid mathematical proof exists and has been verified through appropriate mathematical review.

11.13 Summary

Future work should improve the engine as computational infrastructure. The main directions are distributed computation, independent verification, public data exports, API access, improved runtime accounting, richer visualizations, and archival releases.

These extensions can make the system more useful and more inspectable. They do not change the mathematical status of the Collatz conjecture. Collatz Engine should continue to present itself as a persistent public computational observatory for finite exploration of the $3n + 1$ problem.

12 Conclusion

12.1 Summary of the System

Collatz Engine is a persistent public computational system for evaluating the Collatz map over positive integers. It computes finite trajectories, records stopping behavior, stores peak values, tracks record events, and reports current progress through a public dashboard.

The system is designed around persistence. Engine state is stored outside the browser so that computation can resume after refreshes, redeployments, crashes, or downtime. The database is treated as the source of truth for current number, total checked values, aggregate records, runtime timestamps, and operational status.

The public interface reports finite computational progress. It is not a proof system. It should be interpreted as a reporting and visualization layer over a stored computation record.

12.2 Computation and Interpretation

The computation performed by Collatz Engine is direct. For each starting value n, the engine applies the Collatz map until the trajectory reaches 1, records the relevant computed quantities, and advances to the next starting value.

The main recorded quantities are:

- total stopping time;
- trajectory length;
- peak value;
- termination status;
- record events for longest trajectory and highest peak.

These quantities are meaningful only within the checked finite range. If the engine has completed and persisted results through a bound B, then the system may support a bounded statement about starting values up to B. It does not support the unbounded statement required to prove the Collatz conjecture.

This distinction is central to the report. Finite computation can be correct, useful, public, and reproducible without being a proof.

12.3 Persistence and Auditability

The report treats persistence as a core design requirement rather than an implementation detail. A long-running public computation should be able to recover its state from durable storage. It

should not rely on a browser session or temporary process memory to define public progress.

Auditability depends on the same principle. Reported values should be traceable to stored state, stored results, record-event history, or documented computation rules. The dashboard may format and visualize these values, but it should not invent them.

The system is more credible when it can answer basic inspection questions:

- Which starting value is next?
- How many values have completed stored results?
- What is the highest checked value?
- Which starting value produced the current longest trajectory?
- Which starting value produced the current highest peak?
- When was durable progress last written?
- What evidence supports the displayed status?

These questions are operational, but they affect the interpretation of the computation.

12.4 Role of the Dashboard

The dashboard is part of the public record of the engine. It shows current status, checked values, aggregate records, throughput, and selected visualizations. It also introduces interpretation risk if labels are imprecise or if visual presentation suggests more than the data supports.

For that reason, the dashboard should use careful language. Terms such as "checked values," "observed record," "finite computation," and "current checked range" are appropriate. Terms suggesting proof or mathematical closure should be avoided.

Visualizations should be treated as views over finite data. They may help readers inspect trajectory behavior, record events, and aggregate patterns, but they do not establish general results about all positive integers.

12.5 Limits of the Current System

The central limitation remains mathematical. Collatz Engine evaluates finite cases. The Collatz conjecture concerns all positive integers. No finite checked range proves the conjecture.

Other limitations are operational. The system depends on implementation correctness, exact integer arithmetic, database consistency, result persistence, recovery behavior, and accurate public reporting. These limitations can be reduced through testing, reconciliation, independent verification, public exports, and better tooling. They cannot be removed by presentation.

A technical report should state these limits directly. Doing so does not weaken the system. It defines what the system is and what its outputs mean.

12.6 Path Forward

The next stage of Collatz Engine should focus on improving the system as computational infrastructure. The most useful extensions are distributed workers, explicit range assignment, independent verifier tools, public data exports, better runtime accounting, and archival releases with stable citation metadata.

These additions would make the engine more inspectable and more useful as a public computational observatory. They should be implemented without changing the central interpretation of the system. The engine explores, records, verifies finite ranges, and visualizes observed behavior under the Collatz map. It does not claim to solve the conjecture.

12.7 Closing Statement

This report documents Collatz Engine as a persistent public computational observatory for the $3n + 1$ problem. It defines the computation, describes the persistence model, outlines the data design, states integrity requirements, and clarifies the limits of finite verification.

The system's value depends on precision. It should compute exactly, store results durably, report progress conservatively, and distinguish finite observation from mathematical proof. That distinction is not an external disclaimer. It is part of the system design.

References

[1] Jeffrey C. Lagarias. The 3x + 1 problem and its generalizations. *The American Mathematical Monthly*, 92(1):3–23, 1985.

[2] Terence Tao. Almost all orbits of the collatz map attain almost bounded values. *arXiv preprint arXiv:1909.03562*, 2019.