International Journal of Advance Research Publication and Reviews

Vol 03, Issue 10, pp 732-737, October, 2026

CASCADEX: DISTRIBUTED SIMULATION ENGINE

S.Vijayalakshmi[1] , S.Sanjay[2], S.Sanjaymani[3], K.Saraneshwaran[4], D.Sudharsan[5]

Department of Information Technology, Bachelor of Technology, Sri Shakthi Institute of Engineering and Technology (Autonomous) Coimbatore-641062

ABSTRACT :

CascadeX is a full-stack distributed simulation platform that models how failures spread through interconnected systems. A software architecture is described as a directed graph in which nodes represent services, databases or infrastructure components and weighted edges represent the dependencies between them. When a node fails, the platform propagates the failure along its outgoing edges, reduces the resources of dependent nodes and records every state change so that the complete cascade can be inspected afterwards. The system is built on Node.js and Express with MongoDB for persistent storage, Redis with BullMQ for queue-based tick scheduling, and Socket.IO with a Redis Pub/Sub adapter for real-time streaming to a React dashboard. A dedicated worker executes the simulation as a deterministic state machine, so identical inputs always produce identical results. A Deterministic Heuristic Engine provides pre-simulation risk prediction, detection of critical nodes and bottlenecks, and post-simulation forensic analysis with remediation recommendations. A chaos mode injects random faults to stress-test the topology, and every tick is stored as a snapshot that supports full replay. Security is provided through JWT authentication, role-based access control, rate limiting and environment-aware CORS. Through its modular, horizontally scalable design, CascadeX helps engineers find weak points in an architecture before they cause real outages.

INTRODUCTION

Modern applications are built from many cooperating services, databases, queues and network routes. A fault in one component rarely stays local: an overloaded service slows its callers, the callers exhaust their own resources, and the failure travels through the dependency chain until a large part of the system is affected. Such cascading failures are among the most common causes of large-scale outages, and they are very difficult to reproduce or study on a live production system.

CascadeX addresses this problem by providing a safe environment in which an architecture can be modelled and stressed. The user creates a project, draws the system as a graph of nodes and edges, assigns each node a resource value, a maximum capacity and a failure threshold, and then starts a simulation. The engine advances in discrete ticks: each tick evaluates which nodes have crossed their failure threshold, propagates the resulting load to dependent nodes through weighted edges, recalculates analytics such as the system health score and cascade depth, and broadcasts the new state to every connected client. The simulation stops automatically when the system reaches a stable state in which no further failures occur.

The platform separates the API from the simulation workload by using a Redis-backed job queue, so heavy simulations do not slow down user management or graph editing. Instead of a large language model, CascadeX uses a deterministic heuristic engine for analysis, because architecture stress-testing needs repeatable, explainable and fast results rather than probabilistic text output.

OBJECTIVE

1. To design and develop a distributed simulation platform that models failure propagation across a directed graph of interdependent system components.
2. To execute simulations as a deterministic, tick-based state machine running in a separate queue worker so that results are atomic, repeatable and independent of the API request lifecycle.
3. To provide real-time visibility of the simulation by streaming every state update to the browser through Socket.IO, backed by a Redis Pub/Sub adapter for horizontal scaling.
4. To implement a Deterministic Heuristic Engine that predicts risk before a simulation and produces a forensic root-cause analysis with recommendations afterwards.
5. To support a chaos mode that injects random faults and a forensic replay feature that stores a snapshot of the graph at every tick.
6. To secure the platform with JWT authentication, role-based access control, rate limiting, input validation and environment-aware CORS, and to make it deployable through Docker and a CI pipeline.

LITERATURE SURVEY

Research on cascading failures shows that the structure of the dependency network strongly controls how far a local fault can spread. Motter and Lai demonstrated that in networks where load is redistributed after a node fails, the removal of a single highly loaded node can trigger a chain reaction that

disables a large fraction of the network. Watts modelled global cascades on random networks and showed that the existence and size of a cascade depend on the connectivity of the graph and on the threshold of the individual nodes, which is the same idea used by CascadeX when it compares a node's resource value against its failure threshold.

Buldyrev and co-authors studied interdependent networks and found that coupling between systems makes them considerably more fragile than isolated systems, since a failure in one layer can propagate to the other and back again. Dobson and colleagues analysed series of blackouts in power grids and described how cascading outages arise from the interaction of load, capacity limits and dependencies, which motivates modelling capacity and threshold explicitly for every node. Newman's work on network structure provides the graph-theoretic measures, such as degree and path depth, that are used to identify hubs and single points of failure.

On the practice side, chaos engineering, introduced by Netflix engineers, advocates deliberately injecting faults into a system to discover weaknesses before they cause customer-facing incidents. CascadeX brings this idea to the design stage through its chaos mode. For the implementation layer, the literature and documentation on message queues, publish/subscribe messaging and token-based authentication guide the design: BullMQ provides reliable Redis-backed job processing, Socket.IO provides persistent bi-directional communication that can scale across instances through a Redis adapter, and JSON Web Tokens provide compact stateless authorization. Together, these works indicate that a useful failure-simulation tool needs a graph model, threshold-based propagation, controlled fault injection, real-time feedback and secure, scalable infrastructure.

METHODOLOGY

CascadeX follows a modular, layered methodology. The requirements were first gathered by studying how failures propagate in service-oriented systems and what an engineer needs in order to explore them: a way to describe the topology, a way to inject a failure, a view of how the failure spreads, and an explanation of why

it spread. The system was then divided into a React front end, an Express API organised into feature modules (authentication, project, graph, simulation, simulation history, AI and admin), a Redis and BullMQ execution layer, and a MongoDB persistence layer.

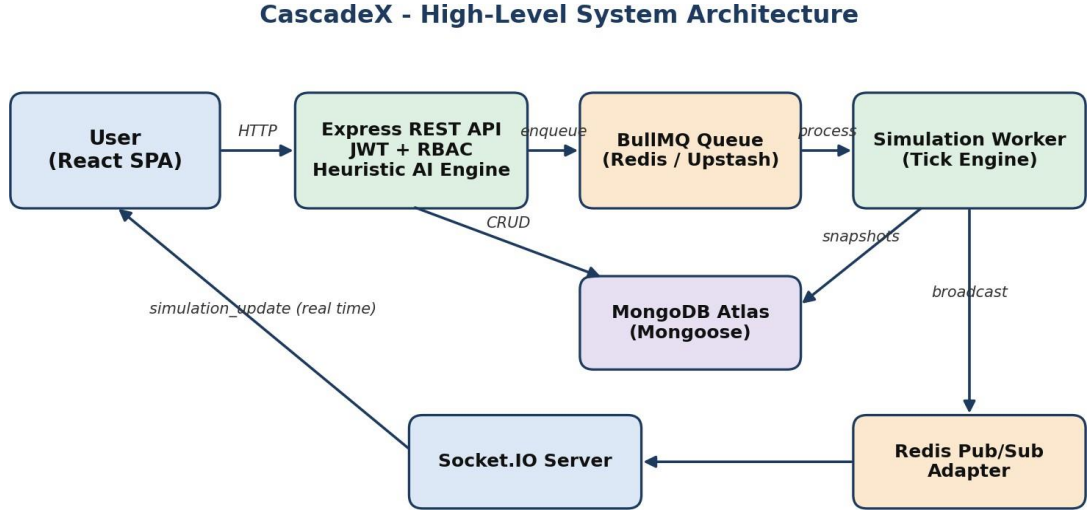
The core of the system is the tick-based simulation lifecycle shown below. The API never runs the simulation itself; it only prepares the initial state and hands the work to the queue.

1. **Initialization:** the user starts a simulation in deterministic or chaos mode. The API acquires a Redis lock (SET NX with a 10-second expiry) so that only one simulation can run per project, loads the nodes and edges from MongoDB and builds an in-memory node-state map and adjacency list. In chaos mode a random node that is still above its threshold is selected and failed.
2. **Queue ingestion:** the initial state is stored in Redis and a tick job is added to the BullMQ queue.
3. **Tick execution:** the worker (concurrency 50) picks up the job and runs three steps: evaluate failures (any node whose resource value is at or below its threshold is marked failed), propagate failures (a breadth-first traversal subtracts the edge weight from every healthy dependent node and fails it if it crosses its threshold), and recalculate analytics.
4. **Persistence and broadcast:** the new state is saved in Redis, a snapshot is written to the SimulationHistory collection, and a simulation_update event is emitted to the project room through Socket.IO.
5. **Completion:** if the tick produced a new cascade, the job re-enqueues itself with a one-second delay; otherwise the simulation stops at quiescence.

The Deterministic Heuristic Engine analyses the graph in two phases. Before a simulation it scores every node using the formula (out-degree × 2) + (in-degree × 1.5) + (sum of outgoing edge weights ÷ 10) + (failure threshold ÷ maximum capacity × 10), detects critical nodes (out-degree above two or outgoing weight above 50), detects bottlenecks (at least two incoming edges and a failure threshold below 30% of capacity), measures the maximum dependency depth with a depth-first search, and classifies the system as LOW, MEDIUM or HIGH risk. After a simulation it finds the structural root cause (the failed node that has no failed predecessor), judges how well the failure was isolated, and generates recommendations such as adding circuit breakers or redundancy for convergence points.

Quality is addressed through a Jest and Supertest test suite for authentication, health and project routes, ESLint linting, and a GitHub Actions workflow that starts MongoDB 7 and Redis 7 services and runs the lint and test steps on every push and pull request. The server is containerised with a Dockerfile and a Docker Compose file, and the production deployment uses Render for the back end and Netlify for the front end.

Figure 1: High-level architecture and data flow of CascadeX



Existing system:

At present, the resilience of a software architecture is mostly judged by manual design reviews, static architecture diagrams and, in mature organisations, load or chaos tests carried out on running environments. Diagrams are static and cannot show how a failure would travel through the dependencies. Chaos-testing tools need a deployed system and can disrupt real users, so they are used late in the life cycle rather than during design. General-purpose simulators and event-driven models are usually built for a single process, are difficult to scale, and may produce results that differ between runs. As a result, engineers often discover cascading weaknesses only after a production incident.

Disadvantages:

- Static diagrams and documents do not show how a failure propagates, so hidden dependency chains remain unnoticed until an outage occurs.
- Fault-injection on live systems is risky, costly and cannot be repeated safely during the design phase.
- Many simulation tools are single-process and event-driven, so they do not scale to large graphs or support several users at once.
- Results are often non-deterministic, which makes it hard to compare two designs or to reproduce a failure for analysis.
- There is little or no real-time feedback, and no recorded history that allows a cascade to be replayed tick by tick.
- Risk assessment is manual or relies on opaque models, giving no traceable explanation of why a component is considered critical.

Proposed system:

The proposed CascadeX platform provides a web-based, infrastructure-agnostic environment in which any architecture can be modelled as a directed graph and stress-tested safely. Simulations run in a dedicated BullMQ worker that processes each tick atomically, so results are deterministic and independent of the API. State is shared through Redis, persisted in MongoDB, and streamed to the dashboard through Socket.IO, which allows the cascade to be watched as it happens. The Deterministic Heuristic Engine predicts risk before the simulation and explains the outcome afterwards with a traceable root cause and recommendations. Chaos mode injects random faults, the Forensics feature stores a snapshot for every tick so that it can be replayed, and an admin dashboard allows platform-wide management. JWT authentication, role-based access control, rate limiting and a Redis-backed lock protect the platform and prevent race conditions.

To illustrate the behaviour of the engine, a sample e-commerce topology of eight components was simulated using the same propagation rules as the server. Each node starts with a resource value of 100 and a failure threshold of 20, and the Order Service is pushed below its threshold. In the first tick the failure spreads from Order Service to Payment Service and Inventory Service, and from Payment Service on to Ledger DB, because each of these receives a weighted hit large enough to cross its threshold. Notification Svc receives a smaller hit (weight 40) and survives. The cascade therefore affects 4 of 8 nodes, the cascade depth is 1, the system health score falls from 100 to 50, and the simulation stops after 2 ticks when no further failures occur.

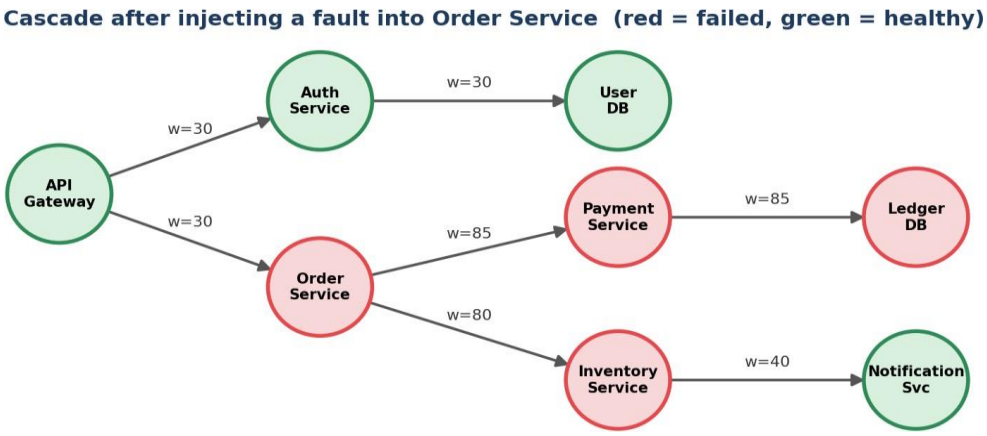


Figure 2: Sample cascade produced by the CascadeX propagation rules

SYSTEM REQUIREMENTS

Hardware Requirements:

- Desktop or laptop computer.
- Intel Core i5 processor or equivalent.
- Minimum 4 GB RAM (8 GB recommended when running MongoDB and Redis locally through Docker).
- 1 GB of free storage space for the project, dependencies and database.
- Internet connection for cloud services (MongoDB Atlas, Upstash Redis).

Software Requirements:

- Front end: React 19, Vite, React Router, Axios, socket.io-client, react-hot-toast, HTML, CSS and JavaScript.
- Back end: Node.js 20, Express 5, Socket.IO with the Redis adapter, BullMQ and IORedis.
- Database and cache: MongoDB (Mongoose ODM) and Redis (Upstash or Redis 7).
- Security: JSON Web Tokens, bcryptjs password hashing and express-rate-limit.
- DevOps and testing: Docker and Docker Compose, GitHub Actions, Jest, Supertest and ESLint.
- Deployment: Render (back end) and Netlify (front end).

Module Description

User Login:

- Allows registered users to authenticate with email and password. The server verifies the bcrypt-hashed password and returns a signed JWT that protects all other routes; login attempts are rate limited.

User Signup:

- Enables new users to register with a name, email and password. Passwords are hashed before storage and every new account receives the default “user” role.

Project Management:

- Lets users create, list, view and delete simulation projects. Each project belongs to its creator, stores its description and keeps the current simulation status (running or stopped).

Graph Builder:

- Allows users to add, edit and remove nodes (name, type, resource value, maximum capacity and failure threshold) and weighted edges between them. A unique index prevents duplicate edges between the same pair of nodes.

Simulation Engine:

- Starts and stops simulations in deterministic or chaos mode, uses a Redis lock to allow one run per project, and executes ticks in a BullMQ worker that evaluates failures, propagates them along the edges and updates analytics such as failed percentage, cascade depth, system health score and most impacted node.

Real-Time Streaming:

- A Socket.IO server with a Redis Pub/Sub adapter places each client in a project room and broadcasts simulation_update events, so that the dashboard shows the cascade live and can resynchronise state after a reconnect.

AI Insights (Heuristic Engine):

- Provides pre-simulation risk prediction (node risk scores, critical nodes, bottlenecks, dependency depth and an overall classification) and post-simulation forensic analysis (injection origin, structural root cause, weakness and recommendations).

Forensics and Replay:

- Stores a snapshot of the whole graph for every tick in the Simulation History collection. Users can browse the stored ticks, replay the cascade step by step, or clear the history.

Admin Panel:

- Gives administrators access to platform statistics, the list of all users and all projects, and the ability to promote users to the admin role, protected by role-based middleware.

The REST API is exposed under /api and grouped by module: /auth (register, login, me), /projects, /graph (nodes and edges), /simulation (start, stop, status), /simulation-history, /ai (predict-risk, analyze-simulation),
/admin and /health. The main collections are shown in Figure 3.

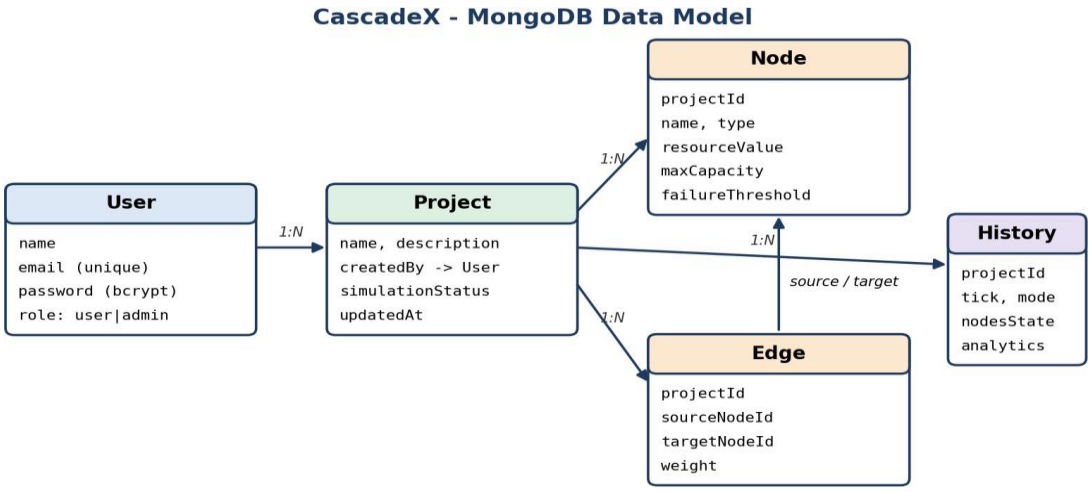
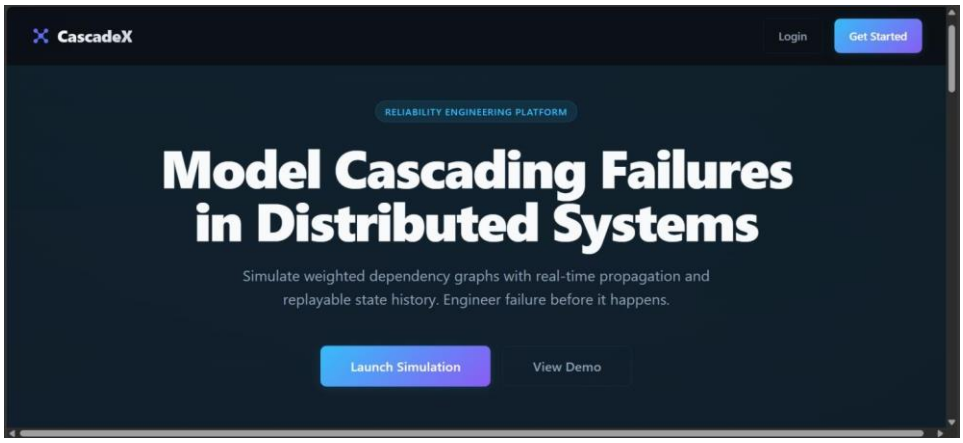


Figure 3: MongoDB data model of CascadeX

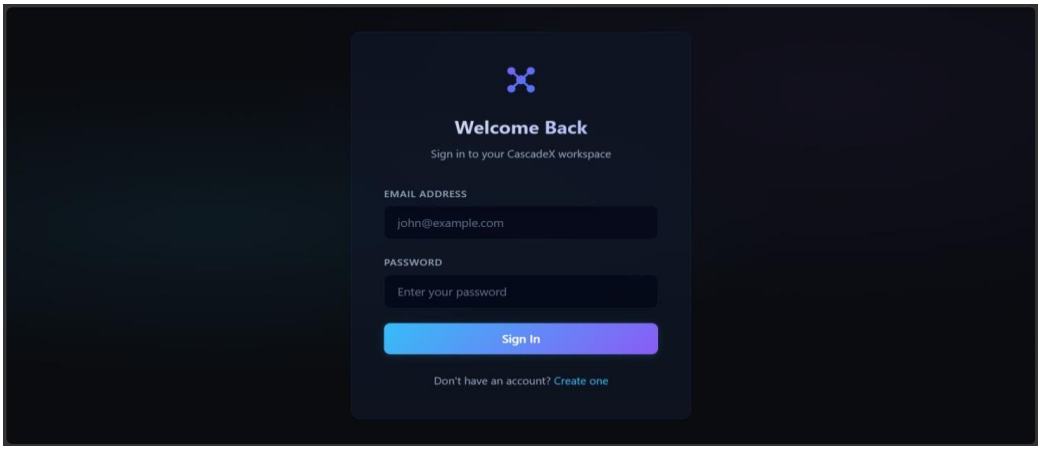
Mode	How the first failure is created	Typical use
Deterministic	A node whose resource value is already at or below its threshold fails first.	Repeatable what-if analysis and design comparison.
Chaos	One random healthy node above its threshold is forced to fail at start.	Stress-testing and discovering unexpected weak points.

Table 1: Simulation modes

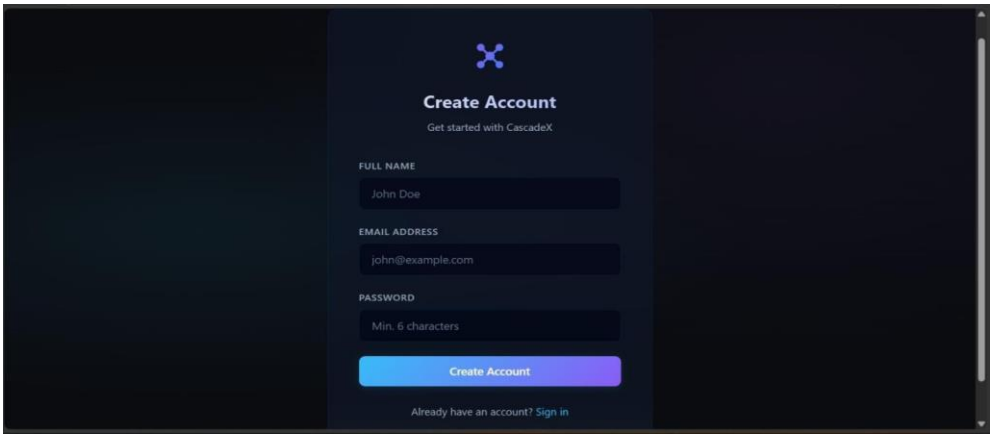
HOME PAGE:



SIGN IN PAGE:



SIGN UP PAGE:



CONCLUSION

CascadeX demonstrates that cascading failures in a complex architecture can be modelled, observed and explained in a controlled environment. By combining a graph model with a deterministic, queue-driven tick engine, the platform produces repeatable results, keeps heavy simulation work away from the API, and streams every state change to the user in real time. The heuristic engine adds explainable risk prediction and post-simulation forensics without the unpredictability of language models, and the stored snapshots allow any cascade to be replayed and studied. Authentication, role-based access control, rate limiting, Redis locking, containerisation and a CI pipeline make the system secure and ready for deployment. Future enhancements include native clustering support for very large topologies, export of graph states as Terraform infrastructure-as-code templates, predictive auto-healing suggestions derived from simulation data, and more extensive automated tests of the simulation engine.

REFERENCES

[1] Motter, A. E., & Lai, Y.-C. (2002). Cascade-based attacks on complex networks. Physical Review E, 66(6), 065102.
[2] Watts, D. J. (2002). A simple model of global cascades on random networks. Proceedings of the National Academy of Sciences, 99(9), 5766–5771.
[3] Buldyrev, S. V., Parshani, R., Paul, G., Stanley, H. E., & Havlin, S. (2010). Catastrophic cascade of failures in interdependent networks. Nature, 464, 1025–1028.
[4] Dobson, I., Carreras, B. A., Lynch, V. E., & Newman, D. E. (2007). Complex systems analysis of series of blackouts: Cascading failure, critical points, and self-organization. Chaos, 17(2), 026103.
[5] Basiri, A., Behnam, N., de Rooij, R., et al. (2016). Chaos Engineering. IEEE Software, 33(3), 35–41.
[6] Newman, M. E. J. (2010). Networks: An Introduction. Oxford University Press.
[7] Jones, M., Bradley, J., & Sakimura, N. (2015). JSON Web Token (JWT). RFC 7519, Internet Engineering Task Force.
[8] BullMQ Documentation. Queues, workers and job scheduling for Node.js on Redis. https://docs.bullmq.io
[9] Socket.IO Documentation. Redis adapter and rooms. https://socket.io/docs/v4/