Maintenance of Large Complex Web-Based Projects

Author: Kamil Rafikov
Email: mailbox@kamil-rafik.com
WhatsApp: +996 755 439 777
Telegram: kamilrafikov

Updated at January 8th, 2026.
Edition: 4 (final fact-checking not yet performed).

Table of Contents

Introduction

The only purpose of this document is to demonstrate for potential employers my knowledge of maintenance of large complex web-based projects. All procedures, cases, and reviews presented further do not relate directly to particular real-world projects. You should apply the provided information to your projects very carefully with rechecking everything in other sources.

Initial, periodic, and one-time operations

Audits

Informal audit procedures presented further are targeted at collecting all information about the project that is necessary for stakeholders and external contractors-maintainers either at the initial stage of collaboration or periodically. They do not aim to reach some regulatory standards compliance. And they do not represent any officially recommended audit framework.

All issues listed in final audit reports must be annotated with severity level, “blast radius”, probability of being encountered, and business impact.

Time estimates for performing audits are given with an assumption that audited system is developed by highly qualified professionals, and all code is rather easily readable.

General system audit

The audit may be performed in the following order:

a) obtain links to all system user interfaces with the appropriate credentials, documentation and knowledge base, issue tracker, repositories;
b) optionally, perform local installation of the project;
c) review all provided information manually, with use of automated code analysis tools, and with use of AI to detect incorrect patterns in implementation and the largest issues;
d) collect information on existing long-term issues and development expectations/plans from representatives of users, from managerial staff, and from technical staff;
e) compile audit report with summary conclusion, 5-20 largest issues, and milestoned strategies to improve the system.

Estimate for a project with several hundred functional components: 3-5 days.

System security audit

The audit may be performed in the following order:

a) agree with all project stakeholders on system decomposition approach;
b) choose threat modeling methodology that fits the chosen decomposition approach maximally (STRIDE, DREAD, PASTA, Tride, Linddun, Attack Trees, CVSS, VAST);
c) define controls and security measures that target identified threats;

d) validate, review, and compile audit report with summary conclusion, 5-20 largest issues, and milestoned strategies to improve the system.

Standard set of searched security issues may include the following from technical point of view (the list is not exhaustive, each system may require different approach).

Client-side issues:

a) users are not trained to avoid standard security issues (e.g. recognizing phishing attacks);
b) certificate management policies on user devices that may allow using malicious certificate authority (as in browser store as in OS store);
c) sensitive browser permissions are requested without real necessity;
d) web interfaces accessed without SSL;
e) absence of CSRF protection;
f) sensitive data stored (or not autoexpired) with cookies, Web Storage API, Cache API, and IndexedDB;
g) absence of two-factor authentication and absence of usage of Web Authentication API;
h) password managers are not used in cases when two-factor authentication is not applicable;
i) publicly available autoincremented/guessable IDs, and IDs mapped to file paths on server;
j) access control implemented on UI level only;
k) risks of XS-leaks attack;
l) risks of clickjacking attack;
m) risks of prototype pollution in Javascript code;
n) compromised NPM packages or external CDN sources of Javascript;
o) NPM packages are not fixed with particular versions.

Web-server, DNS, proxy server, and CDN issues:

a) proxy server settings and CDN settings that cause caching of sensitive data;
b) incorrect CSP and CORS settings;
c) possibility of subdomain takeover;
d) non-prepaid domains and domains not extended automatically;
e) misconfigured DNS that allows phishing emails;
f) misconfigured TLS.

Backend software issues:

a) optionally, absence of fingerprinting of user devices with detection of anomalies;
b) non-encoded output allowing XSS in client-side;
c) dependencies are not fixed with particular versions;
d) account enumeration through too verbose error messages;
e) session ID fixation on identity/role change;
f) keeping session upon change of credentials;
g) incorrect or messy RBAC/ABAC implementation;
h) mass-assignment vulnerabilities;
i) external data validation vulnerabilities;
j) SQL-injection vulnerabilities;

k) over-exposure of data in APIs;
l) insecure deserialization;
m) insufficient rate-limiting in APIs;
n) insecure JWT management (unnecessary data encapsulated, rotation is not applied);
o) insecure integrations with external services;
p) no flags service is used to hide experimental non-polished features;
q) absence of bot protection on business logic layer;
r) weak logout without full session cleanup;
s) absence of idempotency keys;
t) insecure file access control.

Data issues:

a) using non-anonymized copies of production databases dumps for local development and on development server;
b) saving unnecessary sensitive data in logs and databases;
c) audit logs are mutable;
d) over-privileged account used to access database;
e) insufficient or incorrect limits applied on data layer (e.g. allowing negative values for financial operations, absence of unique indexes on records for one-time usage).

Backend infrastructure issues:

a) absence of password and SSH keys rotation policies;
b) password-based access between elements of backend infrastructure instead of using SSH keys;
c) absence of zero-trust networking in backend infrastructure;
d) outdated versions of languages and platforms used for backend development;
e) absent or incorrectly configured application of security updates to all elements of backend infrastructure;
f) using widespread tools for local and server-side development (e.g. Docker) that need specialized complex configuration to be considered as secure and pass regulatory audit for finances-operating systems;
g) absence of alerts on massive consumption of resources as on standalone servers as in the cloud;
h) access to UI of DevOps tools of backend infrastructure without two-factor authentication;
i) insecure storage of passwords and SSH keys by DevOps/development teams;
j) sensitive data embedded into Docker images or settings of CI/CD;
k) insufficient access control in CI/CD tools;
l) using messengers and emails to exchange with credentials and SSH keys in internal communication;
m) access to elements of backend infrastructure available without VPN;
n) no compliance to GDPR;
o) sensitive data backups are not encrypted and RBAC not applied;
p) leakage of data through analytics and monitoring tools;
q) no tested disaster recovery procedure.

Development issues:

a) no unified security standards are enforced across all team members;

b) no code reviews;
c) security audit is delegated completely to responsibility of development team.

All security issues should be searched as in production deployment, as in staging/development/testing deployments.

If you are interested in knowing more details you may find good explanations of some of aforementioned issues in the official MDN and OWASP security guidebooks.

Some tools that allow to automate various security audit tasks are provided further.

CLIENT-SIDE ISSUES	**WEB-SERVER AND DNS ISSUES**
OWASP ZAP Burp Suite Mozilla HTTP Observatory Security Headers by Snyk Retire.js Snyk	SSL Labs Server Test OWASP Amass Subfinder DMARC record checker Cloudflare Security Center
BACKEND SOFTWARE ISSUES	**DATA ISSUES**
Semgrep SonarQube Snyk Code OWASP Dependency-Check 42Crunch Postman Security Schemathesis	TruffleHog GitLeaks AWS Macie Google Cloud DLP Open Policy Agent Nightfall AI
BACKEND INFRASTRUCTURE ISSUES	
ScoutSuite Pacu Prowler Trivy Kube-bench + Kube-hunter	Checkov tfsec Terrascan GitHub Advanced Security Falco

Estimate for a project with several hundred functional components: 5-20 days.

System stability audit

The audit may be performed in the following order:

a) collect feedback from representatives of users, from managerial staff, and from technical staff on outages, performance issues, and past/expected peak usages;
b) configure external analytics and monitoring tools to collect technical reports on current system behavior and analyzing historical data from logs;
c) implement performance data collection on business logic layer in backend software;
d) collect performance limits from user devices and from backend infrastructure;

e) recheck and reconfigure infrastructure alerts on critical issues;
f) recheck log retention strategy;
g) analyze business field to detect possible bottlenecks;
h) perform code review to detect possible bottlenecks;
i) design or improve/upgrade performance testing plan;
j) compile audit report with summary conclusion, 5-20 largest issues, and milestoned strategies to improve the system.

Estimate for a project with several hundred functional components: 3-10 days.

Technological and business legacy audit

The audit may be performed in the following order:

a) collect feedback from representatives of users, from managerial staff, and from technical staff on business logic deficiencies and technical deficiencies;
b) perform code review and infrastructure review to confirm provided information and search missed issues;
c) map and prioritize so-called “legacy pain” for all system staff;
d) determine and document so-called “business logic rigidity points”;
e) determine and document data and business logic aspects that lost meaningless over time;
f) determine and document unused and obsolete parts of code and infrastructure;
g) document informal knowledge on the most critical issues;
h) determine strategy for every part of legacy (rebuilding, isolation, keeping as is) with estimating costs, risks, and horizons of risks;
i) compile audit report with summary conclusion, 5-20 largest issues, and milestoned strategies to improve the system.

Estimate for a project with several hundred functional components: 3-20 days.

Data integrity audit

The audit may be performed in the following order:

a) collect feedback from representatives of users, from managerial staff, and from technical staff on data loss, data duplication, and data inconsistencies;
b) document data integrity expectations to agree where strong integrity and where eventual/delayed integrity are necessary;
c) document fixes that are performed on live data in production environment, also document non-trusted data sources;
d) perform code review to confirm described cases and determine that issue is located exactly in data domain but not in business logic, in client side, or in caching at middleware;
e) upon determining that source of issue is exactly on the data layer then research such aspects as transaction isolation levels, existing/expected separation of data in read/write scenarios, synchronization gaps between data sources used in the system, and caching at data layer;
f) if only single database is used and redistribution of data is not expected then building plan for normalization and indexing may be performed with enforcement of data integrity rules in unique indexes, foreign keys, and null rules;

g) if data are distributed, denormalized or expected to be managed in such way in the future then combination of such techniques may be planned
   - running regular integrity checks (online or offline) with reports/notifications for users and technical staff,
   - automated synchronization and fixing of data integrity,
   - correction of data integrity on business logic or presentation layers with keeping inconsistencies at data layer,
   - versioning of data records;
h) in all cases, history of data changes may be planned for implementation if it is not done yet;
i) in all cases, data integrity monitoring metrics may be planned for implementation if it is not done yet;
j) protocol for manual fixes of data may be needed if such situations are considered as inevitable or as the simplest solution;
k) compile audit report with summary conclusion, 5-20 largest issues, and milestoned strategies to improve the system.

Estimate for a project with several hundred functional components: 1-10 days.

Documentation audit

The audit may be performed in the following order:

a) collect feedback from representatives of users, from managerial staff, and from technical staff on existing and desired documentation;
b) if even basic critical documentation is absent it may be collected immediately and attached to audit report; such documentation may include
   - local installation instruction,
   - release instruction,
   - vaulted list of credentials to used external system and internal administrator-level credentials in the system,
   - main system requirements and metrics,
   - instruction for basic manual testing of primary system functionality,
   - history of the most critical emergencies and methods of their resolution,
   - contacts of all key team members,
   - standard instruction for restoring the system after failure,
   - locations of all system logs;
c) if no information on documentation is provided from project staff then pay attention to presence and quality of such documents as
   - general functional and technical specifications,
   - low-level specifications of internal and external APIs,
   - input/output formats of data for the largest functional blocks,
   - list of possible bottlenecks and technical/organizational issues,
   - logical and physical data storage schemas,
   - general interaction/inheritance diagram of modules and components with only main relationships rendered,
   - numeric limits for different metrics of the system,
   - development standards,

- introductory manual for end-users;

d) formalize procedure of documenting informal knowledge;
e) formalize procedure of embedding knowledge from issue tracker into documentation;
f) formalize procedure of validating documentation correctness and consistency over time;
g) check absence of security-sensitive data in documentation;
h) estimate correctness of applying AI tools for generation of documentation from multiple sources;
i) compile audit report with summary conclusion, 5-20 largest issues, and milestoned strategies to improve the system.

Estimate for a project with several hundred functional components: 3-5 days.

System updates

Preparation for migration

Further the procedure of writing a plan for massive data migration and massive infrastructure migration is covered:

a) check presence of recent data/configuration backups and validate restoration from backup;
b) implement separate production deployment environment to have separate copies of migrated and non-migrated systems; switching between them should be transparent for end-users;
c) if migration of data may cause massive conflict in case of switching users back and forth several times then custom utility tools for resolving conflicts must be prepared;
d) arrange migration procedure for the lowest load hours; if the system services geographically distributed audience, or if this audience may be divided in some another way by business rules then different segments of audience may be migrated separately; separate deployment environments may be needed for every segment in such case;
e) if switching between migrate and non-migrated copies of the system back and forth may last for several days then a procedure covering conflicts resolution in such case may be developed;
f) communication plan for project staff, reservation of additional team members in case of absence of critically important staff, and success checklist may need to be prepared.

It is supposed that automated rollback of massive migration in production system may be rather complex or non-feasible task. That’s why it may be better to concentrate at creating several working deployments of different versions, and on creating utility tools to resolve conflicts between versions.

The tasks of scaling, integration with external system, upgrade of technical environment, implementation of new business processes, and optimization may be considered as subtypes of migration task from technical point of view. That’s why they will be reviewed in this section with providing minor details that are specific for each of them.

Scaling

a) collect from all stakeholders and users exact numeric limits for new system production environment;

b) perform analysis whether scaling may be implemented within the current architecture or complete redesign is necessary;
c) implement procedure, environment, software, and data for preliminary performance testing with new limits.

Integration with external system

a) assess whether some critical latencies will be present and whether asynchronous communication is necessary;
b) assess amount and lifecycle of new data that will be introduced into the system; whether they may be taken as is in raw machine formats, or some abstraction layer is necessary.

Upgrade of technical environment

a) assess amount of unit testing and integration testing to be developed to match new environment; for OS level updates it may be close to zero; for language level updates it may be overwhelming.

Optimization

For optimization the necessity and applicability of the following updates may need to be reviewed:

a) implementation of asynchronous communication between system modules;
b) minimization of hard writes to data storages as much as possible, and keeping them for critical operations only;
c) separate reading and writing to allow independent optimization and scaling;
d) using materialized views for reading complex sets of data;
e) applying real-time OLAP systems to your data infrastructure;
f) applying data lake layer above multiple diverse data sources;
g) applying service mesh layer to multiple microservices;
h) improving communication between microservices with such approaches as avoiding repeated calls to failed microservices, implementing specialized algorithms for repeated calls, isolating failed parts of the system from others to avoid total collapse;
i) implementing zero-trust networking policy;
j) migrating secure operations into trusted execution environments;
k) applying automated terraforming approach to all cloud environments;
l) using automated compliance check if such compliance is necessary in the system;
m) standardization tracing of all operations in the system;
n) applying predictive QA trained on logs data and artificial simulation of failures;
o) migrating to trunk-based development approach for building features that should be delivered immediately;
p) implementing progressive delivery with feature flags to minimize influence of new releases on working parts of the system;
q) implementing not only single local development environment but multiple ones for different versions of the system.

Estimate for a project with several hundred functional components: 3-10 days.

Prevention of critical issues

All critical issues may be resolved with rather simple infrastructure changes and development/deployment procedures:

a) implement reserve production deployment for the system; for the audience which is stratified geographically or according to business rules, several reserve deployments may be necessary; switching between deployments need to be transparent to allow comfortable work upon issues arisen in new releases/bugfixes;
b) implement flag service to allow gradual delivery of new features to the audience;
c) implement regular backup of data/configurations with continuously checked restoration procedure;
d) all backend infrastructure must be monitored with alerts for DevOps on exceeding thresholds of the most important system metrics; responsibility for every type of alert must be documented;
e) in case if system load is variable on temporal scale the preliminary load testing is necessary;
f) implement feature or utility tools to rollback multistep operations and mass operations at once;
g) periodic chaos testing may be necessary;
h) for extreme circumstances DevOps must have a workaround for security policies to allow developers to login into deployed isolated container for performing manual hotfixes or data manipulations;
i) training AI model to detect source of issues with simulating different cases in staging environment.

Ongoing operations

Monitoring of production environment

All project stakeholders, users, and maintainers should come to the common definition of “healthy system behavior” and configure monitoring according to this definition.

Correlation IDs must be implemented to trace propagation of requests across the system and linking all collected metrics to particular IDs.

Correct propagation of request across the system should be visualized to compare monitoring results to the ideal system behavior.

The procedure of monitoring includes configuration of cloud alerts on multiple exceeded thresholds, subscription to alerts from official news channel of the cloud service, installation of SNMP tools on physical servers to obtain raw technical data. Collected metrics may be classified as server hardware metrics (CPU, memory, disks input/output, network communication), application metrics (latencies, error summaries, queues performance, log sizes, deployments, backup operations), business metrics (transactions, user activities etc), security issues (access control anomalies, escalation of privileges, unexpected changes in configurations).

Estimate for development of a monitoring plan for a project with several hundred functional components: 3-10 days.

Code review (as manually written code as AI-generated code)

In modern development environment code review may be performed partially by AI. All developers must either use AI in agent mode or submit completed pieces of code to AI for review. (Only ChatGPT 5 has been checked by me for this purpose, but I have read in some reports that Claude provides better results.) Only part of AI review notes must be implemented (because AI may require too clean code sometimes which is not necessary). Upon completion of AI review the human review must be performed to detect missed issues particularly in business domain and security.

Now let's talk about review of AI-generated code. I have already experience of reviewing such code, that's why it may be stated that you should pay the maximum attention to the following details:

- exact match to business requirements (deep immersion into logic is mandatory, just checking that "all looks logical" is not enough);
- legal cleanliness (e.g. absence of GPL dependencies in commercial project);
- using correct constructions of languages and frameworks, but not just those that work (e.g. check that functions generating true random numbers are used but not functions generating pseudo-random numbers);
- consumption of resources is within expected limits;
- stylistic match of generated code to other parts of software.

Estimate for a project with several hundred functional components: 0-10 hours per week.

Architectural guidance

In modern development environment architectural guidance for project maintenance may consists of the following:

- requiring and checking active correct usage of design patterns; considering participation of AI in review (or even in writing code), active usage of design patterns should not be anymore a blocker causing too much delays in project development;
- tracking low connectivity and well defined boundaries between modules for a case if rebuilding monolithic application with microservices is expected;
- checking legal aspects of used external tools, algorithms, and libraries;
- design and development of draft versions of microservices to be filled further with functionality;
- participation in choosing technological stacks;
- design and tracking development of internal and external APIs;
- keeping architecture decisions documentation.

Estimate for a project with several hundred functional components: 0-10 hours per week.

Gradual refactoring and documenting

The following notes may be considered as a general refactoring/documenting guideline for a case if there is no strict goal to refactor the system but keeping large technical debt is not desired:

- refactoring within several methods should be performed on the fly while developing other features;
- regrouping between modules just for visual separation may be completely enough sometimes;
- agree on some fixed number of lines in a class when refactoring of it becomes obligatory (e.g. 1000-1500);
- aforementioned opportunistic refactoring should not change externally observed behavior if it is not approved by management and covered with tests; however, minor usability improvements may be accepted;
- non-trivial refactoring tasks should be recorded in documentation or issue tracker immediately upon being discovered;
- writing documentation in Markdown files stored in repository may be better than living without documentation at all, if writing it in more readable formalized format takes too much time from developer's point of view.

Estimate for a project with several hundred functional components: 0-10 hours per week per developer.

Disaster resolution cases

Only the most widespread possible reasons of issues are provided further.

Regular outages of the system

POSSIBLE REASON	PREVENTION
Validation of certificates in local client-side store fails because of inaccessibility of some certificate authorities/intermediaries, or proxy inserting non-trusted CA, or user device clock failure	Continuous monitoring of CA trust chain with alerts. Centralized monitoring of user devices settings in organization by system administrator.
System is accessible in fact but UI is invisible due to CSS or Javascript failure: bundle loading failure, bug in code.	Implementation of smoke testing and acceptance testing. Versioning of assets may assist in issue resolution.
Proxy server, CDN, or DNS issue: incorrect routing, regional failure, cache failure.	Continuous monitoring of servers state with alerts (including check from several regions).
Misconfiguration of web-server: incorrect redirect, exceeded limit of connections, incorrect server scripting language configuration.	Continuous monitoring of web server state with alerts on updated configuration and exceeded thresholds.
Process manager failure (e.g. PHP-FPM).	Automated health check and automated memory leak detection with alerts.
API rate limit exceeded.	Continuous monitoring with alerts on exceeded threshold.
Backend software issue: bug in code, lack of memory.	Implementation of unit, integration, and performance testing. Implementation of logging

	and alerts on logs data.
Failed dependency on external service or API.	Implementation of unit and integration testing. Implementation of logging and alerts on logs data. Implementation of health check of external services with alerts.
Database issue: transaction deadlock, lack of memory, exceeded limit of connections, incorrect indexes, incomplete switching between cluster nodes causing remaining node to stay in read-only mode only.	Implementation of performance testing with real data. Relaxed isolation levels for high performance system. Continuous monitoring of database servers state and configurations updates with alerts.
Server issue: disk is full with logs, broken hardware, lack of memory, unrelated software consuming CPU/memory.	Continuous monitoring of server state with alerts on multiple metrics listed in reasons.
Time skew between nodes in backend infrastructure causing authentication failure.	Implementation of mandatory synchronization of time between nodes.
Hacking: DDoS attack, injection of malicious code through non-validated input.	Performing continuous security audit of the system. Monitoring DDoS attempts with alerts. Implementation of unit testing.

Regular loss of business-critical data

POSSIBLE REASON	**PREVENTION**
Data are present but invisible due to CSS or Javascript issue.	Implementation of acceptance testing.
Data are present but missing part of data are processed by separate API or business logic module that fail periodically.	Implementation of integration and unit testing. Implementation of logging and alerting on errors in application logs.
Data are present but soft-deleted.	Implementation of UI for reviewing deleted records.
Data are present but incorrect version is requested if versioning is implemented.	Implementation of UI to choose version of rendered data, or auxiliary UI that shows all other versions automatically (e.g. in tooltip).
Data are present but SQL query is incorrect.	Implementation of integration and unit testing. Implementation of logging and alerting on errors in application logs.
Data are present but not accessible due to RBAC/ABAC rules.	Implementation of integration and unit testing. Implementation of logging and alerting on errors in application logs. Reviewing RBAC/ABAC settings for correctness, clarity, and bugs in configuration.
Data are not synchronized due to database	Implementation of monitoring of replication delay.

replication issues.	Configuring replication to be consistent for critical data.
Data are periodically overwritten by SQL UPSERT query.	Implementation of integration and unit testing. Implementation of logging and alerting on errors in application logs.
Data are periodically overwritten because of concurrency issues.	Implementation of performance testing with real data. Implementation of versioning of data records. Implementation of locks in business logic layer by data IDs or user IDs.
Data are rolled back after writing due to incorrect transaction boundaries.	Implementation of integration and unit testing. Implementation of logging on transaction boundaries. Implementation of strict review procedures for transaction boundaries.
Data are periodically deleted by scheduler job, or manually executed console script, or asynchronous post-processing.	Implementation of integration and unit testing. Implementation of logging and alerting on errors in application logs.
Data are not persisted because of crashing scheduler jobs.	Implementation of logging and alerting on crashes of scheduler jobs.
Data are periodically deleted by incorrect business logic.	Implementation of integration and unit testing. Implementation of logging and alerting on errors in application logs.
Data are deleted manually in UI by some users.	Implementation of logging and alerting on errors in application logs. Implementation of stricter rules on data layer. Implementation of new RBAC/ABAC rules.
Data are deleted as a part of GDPR procedures.	Division of data covered by GDPR and data that should not be deleted.
Data are separated for reading and writing, and synchronization does not occur.	Implementation of integration and unit testing. Implementation of logging and alerting on errors in application logs. Separation of parts of data that are persisted immediately.
Data never persist, because of being calculated and stored in frontend or temporary cache storage only, and then removed.	Implementation of integration and unit testing. Implementation of logging and alerting on errors in application logs. Tracking lifecycle of critical data if they need to be stored in non-persisted form for a long period of time. Separation of parts of data that are persisted immediately.
Data persist, but triggers remove them.	Documenting triggers if usage of triggers is necessary. Generating audit log records from triggers.
Dependency on external system that changed its	Implementation of integration and unit testing.

behavior caused issues in data management.	Implementation of logging and alerting on errors in application logs.

Regular duplication of business-critical data

POSSIBLE REASON	PREVENTION
Data are unique in storage but duplicated on client-side only because of Javascript issue.	Implementation of acceptance testing.
Data are unique in storage but duplicated because of incorrect SQL JOIN query.	Implementation of integration and unit testing. Implementation of logging and alerting on errors in application logs. Using GROUP BY and DISTINCT in all such queries.
Different versions of the same data are rendered.	Implementation of integration and unit testing. Implementation of logging and alerting on errors in application logs. Filtering out unnecessary versions on business logic layer or on presentation layer.
Client-side bug in Javascript allows double click on data saving operation.	Implementation of acceptance testing. Implementation of unique index in database. Implement idempotency keys. Implementation of unified saving logic on client side that locks UI until getting server response.
Duplicated HTTP request is coming to server on data saving operation.	Implementation of unique index in database. Implementation of idempotency keys.
Background jobs process the same data multiple times.	Implementation of idempotency keys for background jobs. Keeping processed data IDs with unique index on them.
Concurrency issue that creates data before uniqueness is enforced.	Implementation of locks on business logic layer for data IDs. Keeping processed data IDs with unique index on them.
Data are coming from duplicate retried events.	Keeping processed data IDs with unique index on them.

Loss of project knowledge base

POSSIBLE REASON	PREVENTION
Development has been performed by outstaffing and documentation was not included into contract.	Documentation (with strict criteria for amount and formats) must be explicitly included into contract with an external agency.
Personal conflict or accident prevented developers to provide documentation.	Documentation must be written as in opportunistic style as with devoting special time for this activity regularly. In result, some documentation will

	always be present.
The system is a legacy product obtained from some vendor and contacts of initial development team are absent.	AI may be used to analyze codebase and provide some insights on non-standard implementations. However, its capabilities should not be overestimated.
Documentation has been accidentally deleted because of technical failure.	Backups should include not only code but documentation too. Ideally, documentation should be stored in the same repository as source code.
Gradual erosion of knowledge base due to continuous rotation of staff.	Implementation of strict mandatory procedures for knowledge transfer upon onboarding. Implementation of minimal critical documentation.
Documentation is present but it is too non-systematic, too verbose, outdated, and it cannot be matched to code easily.	Implementation of external review procedure for documentation at least once per 1-3 months.

Long-term maintenance issues

Besides disaster cases described above it may be possible to encounter other issues that accumulate gradually over long periods of time. Here is just a short summary of such issues that cannot be resolved easily and may require unique approach:

- architecture development reaching evolutionary limits;
- too tight connection to vendors of software tools and infrastructure;
- impossibility to meet new regulatory requirements due to large amount of old solutions;
- extremely high cost of onboarding of new team members;
- impossibility to implement zero-trust internal networking policy due to high connectivity of system internals;
- too much “temporary” features, flags, solutions, data etc;
- non-manageable amount of historical data.

Summary

I hope this document demonstrates my capability to maintain large complex web-based projects, I hope you enjoyed reading, and I expect long-term productive work with you. Thank you for spending your time!