Technical Environment Operations for Large Complex Web-Based Projects

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

Updated at January 15th, 2026.
Edition: 3 (final fact-checking not yet performed).

Table of Contents

Introduction

Even in relatively large web-based project the range of tasks covered by such terms as development operations, security management, and IT Operations may be performed just by 1-3 persons and combined with software development activities. That's why I prefer to use the term *technical environment operations* to describe this range of tasks. Usually, such tasks are not in my responsibility, however I have decided to compile this checklist for potential employers for a case if I will be ever assigned to all this work to allow them to know what must be done and to demonstrate limits of my qualifications in the described topics. All provided information may be incomplete. You must recheck it in other sources if you have decided to apply the checklist to your projects. Topics are covered from the most important ones to less important ones from the point of view of typical end-user and business stakeholder.

Healthy system

Instead of concentrating on technical aspects, I would like to put at the first place the term *healthy system* to highlight such details as complexity of modern systems which is close to biological ones, active usage of AI for resolving multiple tasks (expected or already introduced usage), and enormous amounts of processed data. I would like to propose the following definition of a healthy system:

a) operating with agreed downtime and with all metrics staying within predefined thresholds even on performance peaks;

b) having reserved resources for accidental increase of performance;

c) keeping all processed data correct;

d) keeping propagation of all requests passing through the system along predefined paths without metrics exceeding predefined thresholds at any point of collection;

e) keeping all latencies within the current performance standards of the project's business domain;

f) providing immediate diagnostic output on any issues and on exceeded thresholds that allows resolving issues or allocation of additional resources before any business impact is present;

g) reducing to the agreed minimum the risk of unauthorized access to data, services, and resources;

h) keeping history of all critical business operations and critical technical operations for audit;

i) allowing immediate restoration in case of force majeure;

j) staying compliant to regulatory standards in project's business domain.

In each section further only the most important monitoring metrics are provided.

General implementation approach

From technical points of view, modern technical environment for a large complex web-based project may be characterized with the following:

- declarative approach to configuration; all environment configuration is stored in Git repository and applied to the cloud in several steps through Infrastructure-as-Code engine and terraforming functionality;
- deep segmentation of environment for granular access control with a tendency to make access control identity-centric;
- multi-layered architecture with tendencies to move latency-sensitive and cost-sensitive operations closer to clients;
- necessity to support legacy protocols, wide range of end-user devices, different software clients, and backend "zoos" of technologies.

Many details provided further describe an ideal system configuration. Real-world situation in old projects may differ much, but may be considered as acceptable one too.

External access

Protocols

Prefer HTTP/3 over HTTP/2 on user-facing traffic where possible, but keep dual support because using the first one only may cause reachability issues in corporate/legacy networks. Rely on TLS 1.3 but keep TLS 1.2 support too.

Monitoring metrics: number of requests per second, route, method, client; request/response sizes; request duration (95^{th}–99^{th} percentile is mandatory to track real user experience); processing time on

different layers of the system; error rates by type; overload detection metrics (active, queued, dropped, retried connections); TLS protocol-level parameters; certificates health metrics.

IP addresses

Support dual stack IPv4 and IPv6. Even now (beginning of 2026), it is still important for global access. Complete migration to IPv6 is expected not earlier than the mid-2030s. Try to avoid considering IP address as identity for access control purposes.

Implement mandatory IPv6 firewalling. (IPv6 eliminates NAT and creates possibility to reach directly and globally any listening service on device with IPv6 address. Large span of addresses does not protect against targeted attack.) Linux security architecture assumes presence of IPv6 firewall, without it multiple services become exposed.

Monitoring metrics: IP address to IP address traffic parameters; reachability, latencies, and packet loss for particular IP addresses; subnets capacities; address assignment stability; number of established, waiting, and failed connections.

Ports

Do not bind custom services to ports <1024, prefer high registered or dynamic ports for this purpose. All open ports must be explained in documentation and monitored. All ports except those that are really needed must be closed, opening any new port should be tracked by automated scanning and cause an alert. Public ports should be present in reverse proxy or gateway service only (and usually it should be 443 port only).

Monitoring metrics: availability, usage, overusage.

DNS

At least 2 authoritative DNS providers must be configured if it is financially acceptable. Anycast DNS (sharing IP address between multiple servers) for maximum performance and security is preferred. Internal recursive resolver should not be exposed for public use. Prefer IPv6 where possible if all is firewalled correctly. Use DNSSEC as standard de facto for all public domains (but keep in mind that it may increase outage “blast radius” in case of mismanagement). If possible, prefer DNS-over-HTTPS or DNS-over-TLS for better security. Strictly prefer usage of cloud provider tools for zones editing instead of manual edits.

Mailing configuration of DNS is not covered in this section because it is not relevant to maintenance of web-based projects usually.

Monitoring metrics: rates of queries per second, zone, resolver; success rate; NXDOMAIN, SERVFAIL, DNSSEC, and validation errors; propagation delays; region-specific latencies

APIs

You should expect from developers the following:

- REST will be used as primary API architecture, GraphQL (for frontend data aggregation) and gRPC (for inter-service communication) will be used occasionally;
- data will be primarily passed in JSON format;
- semantics of requests will strictly match HTTP methods;
- versioning will be mainly implemented with routes prefixes in such style as */v1/*.

Monitoring metrics: number of requests per second, route, method, credential; payload size; distribution of latencies by system layers; error rates by type.

Performance

CDN

For large-scale systems CDN must be considered as a mandatory edge layer due to the following reasons: better static assets delivery, speeding-up TLS procedures, caching API responses, DDoS protection, performing some technical computations (routing, feature flags etc) at edge layer but not in application layer.

Monitoring metrics: cache hit ratios, error rates, geography-based latencies.

Load balancing

Load balancing may be applied to the system on multiple layers. CDN by itself functions as a part of this multi-layer load balancing subsystem. (In some types of systems implementation may shift gradually from centralized architectures to client-side load balancer choice.) Algorithms used by load balancer should be based on collected statistics of latencies, load, and errors, but not just rotating nodes. Also, some system protection functions are moved to load balancer level.

Monitoring metrics: number of requests, retries, and connections per second and node; latencies on multiple system levels; error rates grouped by types; CPU, memory, filesystem usage levels.

Autoscaling

Autoscaling logic may cover not only servers, but penetrate deeper up to application level (services, workers, particular features). Ideally, operation of autoscaling should be predictive and be based on combination of multiple signals from different layers of the system, in other words, it must be as business-oriented as hardware-oriented. (As you can guess, the real world situation may differ often.) Resources usage may be extended into three dimensions step by step: a) firstly, with adding more data replicas (in fact may be added in advance); b) then, with adding more nodes/workers/threads; c) and

finally, with increased power of particular server instances (less preferred option). Safety limits for usage of resources with alerts are mandatory. A policy of conscious gradual system degradation may be applied with a purpose to avoid total failure. Modern autoscaling implementations are primarily Kubernetes-based.

Monitoring metrics: number of requests, retries, and connections per second, node, and service type; latencies on multiple system levels; error rates grouped by types; CPU, memory, filesystem usage levels.

Deployment

Kubernetes

Kubernetes should be considered not as a tool but as cloud-native OS. Technical operations team's task is to build multi-layered platform on the top of Kubernetes' native functionality: base cluster, security settings, monitoring settings, CI/CD settings, UIs and APIs for developers. Settings management should be performed via Git repository for automation and history tracking purposes.

Technical team should check that development team designs application with keeping Kubernetes features and architecture in mind. This approach includes such techniques and patterns as serverless services, state externalized in databases and queues, obligatory health checks of services, graceful shutdowns, impossibility of SSH access, cron jobs managed by platform, automated restart, security management on platform level but not on service level, complex structured tracing output.

However, no Kubernetes is necessary for systems consisting just of several simple services operating with minimal latencies.

Monitoring metrics: all collected metrics as latencies, requests, errors, hardware usage should be as on services layer as on node level; all collected metrics should target autoscaling management primarily but not just be kept for history or manual review.

Containerization

Main principles of containerization include the following: immutability (no SSH, no packages updates), versioning, rebuilding for fixes, minimal base images, read-only root file system, externalized configuration, possibility to use container on multiple runtimes (e.g. no strict dependency on Docker), Kubernetes-managed orchestration in production runtime, configuration managed via Git repositories as a single source of truth, stateless (diskless) implementation, compatibility of local development settings and production settings. Security aspects include mandatory image scanning, image signing, rootless implementation, dropped Linux capabilities, access control systems built-in.

Possibly, Docker should be avoided in production environment because of its well-known security problems and typical development usage patterns that will cause a lot of issues upon being moved to production by inexperienced development or technical teams. Podman (alongside with Buildah) is a

recommended replacement for Docker (but there are several tens of other tools that may be considered too). Speaking frankly even in not too complex Docker-centric local development environments Podman may be chosen too as a replacement. (Certainly, if your DevOps is “a Docker guru” and can devote some time to assist developers in local installations management then you may stay in Docker world.)

For Kubernetes-centric development you may try also such tools as K3s/K3d. If you need full OS containers LXC/LXD may be tried.

Monitoring metrics: CPU, memory, filesystem, network usage; startup, restart behavior; deployment failures; security issues such as escalations of privileges, unexpected network connection attempts, changes in processes count.

Fallback production deployments

You should always expect that failures are inevitable. That’s why in an ideal environment there is no such thing as fallback production deployment. Instead of this a gradual delivery of new versions to the audience segmented geographically or by business logic rules should be implemented. However, for simpler system and minimal budgets old-fashioned approach with traffic switch control between two deployments may be acceptable solution too.

Feature flags service is rather simple fallback solution that may be developed from the very beginning of every large project.

Database migrations rollbacks as a fallback solution is acceptable for the simplest pieces of logic only. Concentrating attention at having separate database for mass migration of data for limited part of audience, and at implementing data consistency restoration tools are preferred options for large projects.

In really large-scale mature systems all fallback solutions should be completely automated with basing on multiple monitoring metrics.

Configuration management

All configurations must be managed by Git repositories in the same way as code. Baking configuration into container images should be strictly avoided. Environment awareness should be performed not through multiple separate configuration sets, but through multilayered configuration loading some settings from environment variables. Secrets management should be performed either at Kubernetes level or at cloud keys management service level (auto-rotation of secrets is mandatory). Configuration changes may have the same importance as critical system errors or exceeded latency thresholds.

Monitoring metrics: manual change attempts, high-risk settings changes (upon classification of settings by risk level), applied configuration version changes, configuration reloads, feature flags switches, secret access failures.

Workaround access for emergency

In spite of paying much attention to such aspects as immutability of containers and inaccessibility of containers by SSH, real world situation may differ and in typical SME you may encounter a situation when only immediate manual fix of business-critical issue in a container deployed to production environment may resolve situation. For this purpose a workaround allowing to do it must be implemented by technical environment team. Access should be controlled by short-lived centrally managed list of SSH keys/certificates to allow developer to enter the system upon adding his public SSH key/certificate into that list. If it is possible, SSH access should be performed not into container itself but into Kubernetes node to keep audit logs.

Recreation and restoration

Terraforming

Terraforming configuration settings must be managed via Git repositories. The procedure of terraforming should not be considered as something critical applied in emergency situation only, but as a normal procedure for deploying experimental features or deploying separate environments for segmented audience.

Backup

Backup policies and settings must be managed via Git repositories. Backup strategy should meet the following rule: data should be stored in 3 copies, 2 different storage types must be used, 1 copy must be stored off-site, 1 copy must be immutable (e.g. in locked object storage), all copies must be checked for restoration automatically and close to 0 errors should be detected in the process of restoration (non-restored backup is non-existing backup). Backups should be performed under separate set of credentials not used in standard system operations. Secure data may require separate backup workflow and encryption.

Further additional particularities of different types of backups are provided.

- Data backup procedure depending on project scale and environment may be performed as by Kubernetes tools, as by terraforming tools, as by native DBMS tools. Data backup structure may be layered in the following way: volume snapshot, database native backup, application metadata.
- Git repositories should be backed up with mirroring tools.
- Documentation should be stored and managed as code, ideally together with code. So, the procedure of source code backup will involve backup of documentation too. If some external documentation management tools are used then backups of regular full exports are mandatory.
- Release artifacts must be backed up too.

- Mission-critical system may practice almost continuous backup with restoration granularity up to seconds but it is not covered in this document.

Monitoring metrics: completion of backups, restoration failures, exceeding backup storage thresholds.

Services

Web servers

In modern large-scale projects web servers are practically ephemeral, they are part of infrastructure created in runtime by configuration stored in Git repository and running in container. Even if your project did not grow up to this level yet and your web server is installed and managed manually the same basic principles are applied to it too. Here are they are: all is under TLS, certificates are rotated automatically, headers management should match well-known MDN/OWASP security guidelines, rate limits are applied by default. Depending on system requirements the web server may play three roles: a) edge proxy, b) reverse proxy, c) application server.

Monitoring metrics: number of requests per second; web server and upstream latencies; error rates by type; connection pool utilization; rate limits thresholds; TLS health; CPU and memory usage.

Database servers

If you moved to the cloud, you should try to use managed database services instead of standalone servers. Database failures should not be considered as standard incidents, it should happen rarely and with a minimal business impact in well-designed, well-managed environment. No public database endpoints should be available, all should be accessible within internal network only. All schema changes should be performed by migrations only.

Monitoring metrics: query latency, number of transactions per second, connection pool utilization, replication lag, lock wait time, disk input-output statistics, cache hit ratio.

Process managers

Process managers configuration and monitoring approach will be reviewed with using PHP-FPM as example. It may be recommended to configure it according to the following principles:

- setup demand-based resources allocation;
- separate pools of processes according to business logic requirements;
- maximum number of processes should be specified manually with strict limit considering real memory requirements and maximum possible amount of auto-allocated autoscaled memory;
- consider long-running workers as containing issues and terminate them automatically;

- optionally, remove from workers all environment variables except those explicitly passed by configuration;
- log all into stdout/stderr where logged data will be caught by centralized cloud logging system;
- do not expose process manager to Internet.

Monitoring metrics: number of active and idle processes; utilization of workers pool; number of waiting requests; latencies; fork and termination rates; CPU utilization per request; distribution of memory usage by workers.

OS-level services

Current approach to OS management in large web-based projects consists of using containerized immutable minimalistic images with keeping them separately from application layer. Security management principles include blocking of password authentication, strictly limited or blocked SSH authentication, enabling kernel security modules for access control, read-only root filesystem.

Monitoring metrics: CPU, memory, and filesystem utilization; disk input-output latency.

Security perimeter

It should be strictly noted that in modern large-scale systems there is no single security perimeter with flat internal access. Microsegmentation with centralized management of rules for accessing every segment is the preferred system architecture approach.

Threat modeling

Threat modeling should be implemented as continuous activity performed in interaction between development and technical environment teams. Every new feature and new configuration introduced into the system should be analyzed from the point of view of security, and such analysis should produce new or update existing security guidelines which finally should result in security-related system reconfigurations. Key aspects of threat modeling include visualization of external entities affecting the system, data flows, trust boundaries, identity and authorization paths, external dependencies. Every detected threat is leveled according to its “blast radius” and business impact. Every prevention measure is reviewed in the context of possibilities given by cloud/platform infrastructure and external tools.

DDoS prevention

In modern technical environment DDoS prevention is delegated to CDN primarily. The second level of protection is firewall configured to handle rate limits adaptively. At the third level, API gateway quotas and aggressive caching protect the system.

Firewall

Identity-aware firewalling is the preferred architecture approach for new systems. IP-limiting only is still necessary but considered as a complementary functionality. As all other in the large-scale system, the firewall rules should be defined as code and managed via Git repository, and then applied through cloud infrastructure tools on multiple layers.

Monitoring metrics: ratio of denied to allowed connections, unexpected traffic amounts.

VPN

VPN does not represent security perimeter, it is just minor part of it. Its primary use is just limiting access to administrative tools. Usage of VPN does not eliminate necessity in two-factor authentication. Even upon access to VPN the team members should be able to access internal resources according to their roles, no flat total access should be given. WireGuard is one of the most widespread tools to build VPN.

Zero trust internal networking

This concept is based on access control based on identities of users, services, devices, but not on IP addresses and subnets. Identity is one of the primary aspects that determines internal abstract networking boundary. All connections are expected to be TLS-based with use of short-lived certificates. Internal network should be microsegmented.

Certificates management

In modern technical environment, certificates are preferred to be short-lived (90 days or less, down to several minutes in technical infrastructure) and automatically rotated regularly. Every service and user in the system is given certificate-based identity. Manual certificates management should be considered as undesired. All internal traffic between services should be encrypted with use of certificates, no shared keys should be used (certainly, such approach will work for new systems only, in legacy systems all may be different). Different CA should be responsible for different trust domains, compromising everything through single CA should be impossible.

SSH keys and certificates

In small-scale projects with minimal budget you may still rely on authentication with use of SSH keys. However, in larger systems there is a trend to migrate to authentication with use of time-limited SSH certificates. These certificates are issued upon identification via identity provider (e.g. hardware token in two-factor authentication), the server trusts CA of certificate but not the key. Authorization rules are applied on user role level but not on keys level.

Two-factor authentication

Hardware tokens must be used for implementing two-factor authentication if it is possible from financial and organizational points of view. TOTP passwords from authenticator app is an acceptable fallback solution in less secure environment. Other options (e.g. SMS, email codes) should not be used in the modern systems anymore.

Management of owner-level credentials

In this section I would like to highlight the process of managing owner-level credentials to the system. Such credentials may give total access to everything, that's why the following requirements must be met in operations involving them:

- two credentials, main and backup, must be used to counteract phishing attacks;
- credentials must be stored either in offline vault or in specialized external service (e.g. HashiCorp Vault);
- access to credentials must require mandatory two-factor authentication with hardware tokens or authenticator apps (biometric authentication may be possible too, but reviewing it is outside of scope of this document);
- access to credentials must require >1 person with keys/tokens (or even also >1 person who approve such access);
- access should produce immediate immutable audit record and alert to all responsible personnel;
- rotation of credentials upon every access is necessary if it is possible;
- login with such credentials should be possible from tightly controlled environment only (e.g. device with fixed IP address).

Management of other credentials

Authentication and authorization should be implemented with the following principles:

- authentication with cloud identity service providing short-lived credentials;
- cloud-native or external services for managing dynamic secrets must be used;
- no root login or root access keys should be present on every node; two-factor authentication is required for root-level access;
- separate administrator roles for all kinds of tasks with minimal sets of permissions;
- no secrets in CI/CD and automation; using cloud tools for access control;
- alerting on all security-related operations.

Endpoint security control in internal team users' devices

If possible, it is desired to implement some level of control over end-users' devices in internal team through regular OS and firmware updates, continuous malware detection scanning, disk encryption, limiting functionality of browsers, accessing devices by accounts with limited permissions, enforcing two-factor authentication, remote locking and wiping out of devices in case of compromise.

Monitoring

Tracing

Developers must use OpenTelemetry libraries as de facto industry standards. W3C Trace Context standard must be used to track propagation of requests through the system. Prometheus may be used as the tool for collecting metrics in cloud environment and (together with Alertmanager) as alerting subsystem.

Logs management

Management of logs from all system layers must be centralized. Logs messages are expected to be structured (e.g. in JSON format) but not one line in plaintext. Writing grep-oriented logs should be considered as bad obsolete practice. Log messages should be correlated to tracing information with inclusion of trace IDs into messages. Application logs and audit logs have different purposes, the primary particularity of audit logs is their immutability, and they should never be mixed. Secure data must be masked in log messages.

Analytics

Generally, analytics is not in responsibility of technical environment team although it may be helpful in incidents resolution sometimes. However, technical team is responsible for configuring event management tools (e.g. Kafka) and data storages (e.g. ClickHouse) to process analytical events. Primary configuration aspects in this case are storage limits, connectors management, structure of topics, search indexing approach; all this must be resolved in tight communication with the development team.

Alerting

Alerting should be primarily based on metrics but not on logs. Group and target alert destinations (technical personnel pagers, internal company messenger, helpdesk system) by levels: a) alerts on exceeding technical thresholds, b) alerts on exceeding business operations thresholds. Every type needs its own responsibility level and range of performed actions.

Knowledge base

All technical environment operations must be documented with the same (or even higher) accuracy as software development. Further, the main types of documents are given to explain what should be expected: initial system configurations, operations responsibility/accountability assignments, system update reports, incident resolution reports, managerial instructions on all technical operations (sometimes coming in informal way in messengers or emails), customer complaints and resolutions of these complaints, personnel onboarding/dismissal instructions, emergency instructions, documentation responsibility/accountability assignments, system credentials passing procedures.

Incident resolution reports must be timestamped and include analysis with business impact estimates.

All this documentation must be versioned and backed up.

Example of basic monitored metrics thresholds

Further, I would like to provide you with a basic example of the most critical monitored metrics for a large-scale web-based project. Thresholds are given to have any idea about what may be expected in the current business environment. You should use them just as a theoretical guidance for an ideally functioning large generic web-based system, no more.

General notes: a) exceeding of thresholds should be consistent over some timespan to be considered as an event, e.g. 5 minutes; b) statistics collected over 95-99% requests/operations matter more than average values; c) correlation of metrics must be analyzed, but not just any single metric.

METRIC	HEALTHY	WARNING	CRITICAL
NETWORK			
Successful / failed requests ratio	≥ 99.95%	< 99.90%	< 99.50%
Response time latency for 95% of requests	≤ 500ms	500-1200ms	> 2000ms
Response time latency for 99% of requests	≤ 1200ms	1200-1300ms	>5000ms
5xx HTTP errors / total requests ratio	< 0.1%	0.1-1%	> 1%
4xx HTTP errors / total requests ratio	< 1%	1-5%	> 5%
Current RPS / tested RPS ratio	≤ 60%	60-80%	> 85%
Pending requests in load balancer queue	~ 0	> 50 sustained	growing
Autoscaling reaction time	< 60s	60-180s	> 180s
HARDWARE			
CPU usage in 95% of nodes	< 65%	65-80%	> 85%
Memory usage in nodes	≤ 70%	70-85%	> 90%
Out of memory events in nodes	0	> 1	regular

Disk space usage in nodes	≤ 70%	70-85%	> 90%
Disk input-output latency in 95% of nodes	≤ 5ms	5-20ms	> 50ms
Filesystem inode usage in nodes	≤ 70%	70-85%	> 90%
DATA			
Latency in 95% of database queries	≤ 100ms	100-500ms	> 1000ms
Used / maximum database connections ratio	≤ 60%	60-80%	> 90%
Data replication lag	≤ 1s	1-10s	> 30s
Oldest unprocessed message in queue	≤ 30s	30-300s	> 10min
Data cache hits / total reads ratio	≥ 90%	80-90%	< 80%
BUSINESS			
Successful / failed business operations ratio	≥ 99.50%	98-99.50%	< 98%
Business domain failures / total business operations ratio	< 0.5%	0.5-2%	> 2%
Idempotency errors / total business operations ratio	0	> 0	regular
Invalid state errors / total business operations ratio	< 0.1%	0.1-1%	> 1%
Business transaction duration	≤ 1-2s	2-5s	> 5s
Complex business workflows completed / started ratio	≥ 99%	95-99%	< 95%
Business invariant violations	0	> 0	regular
Time to reach consistency over subsystems	≤ 1-5s	5-30s	> 1min
Number of business operations per second	predictable	drop/peak	collapse
Max / average usage ratio by single customer	≤ 10x	10-50x	> 50x
Mismatch of financial records	0	> 0	regular
Financial rollbacks / total transactions ratio	< 0.5%	0.5-2%	> 2%
Number of authorization failures	predictable	drop/peak	collapse
Post deployment business errors delta	≤ +0.2%	+0.2-1%	> +1%

Summary

I hope this document demonstrates my capability to deploy and 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!