**ARCHI.Tech Hackaton**

**«Архитектурный стандарт надежности информационных систем»**

Команда: Beverly Hills Misis

**Москва, 2025**

Оглавление

[**Введение** 3](#_Toc198829592)

[**Глава 0: Надёжность физической и сетевой инфраструктуры** 3](#_Toc198829593)

[**Глава 0.1: Инженерная надежность** 3](#_Toc198829594)

[**Глава 0.2: Сетевая архитектура** 4](#_Toc198829595)

[**Глава 1: Культура разработки** 6](#_Toc198829596)

[**Глава 1.1: Отказоустойчивая разработка** 6](#_Toc198829597)

[**Глава 1.2: DevSecOps практики** 7](#_Toc198829598)

[**Глава 1.3: Безопасная инфраструктура (IaC)** 9](#_Toc198829599)

[**Глава 2: Архитектура кластера** 10](#_Toc198829600)

[**Глава 2.1: Топология и модели размещения** 10](#_Toc198829601)

[**Глава 2.2: Сетевая интеграция и балансировка нагрузки** 11](#_Toc198829602)

[**Глава 2.3: Оркестрация и жизненный цикл** 14](#_Toc198829603)

[**Глава 2.4: Управление обновлениями** 17](#_Toc198829604)

[**Глава 3: Хранение и передача данных** 19](#_Toc198829605)

[**Глава 3.1: Уровни хранения** 19](#_Toc198829606)

[**Глава 3.2: Репликация и кворум** 22](#_Toc198829607)

[**Глава 3.3: Интеграция и согласованность данных** 24](#_Toc198829608)

[**Глава 4: Мониторинг и управление инцидентами** 25](#_Toc198829609)

[**Глава 4.1: Метрики и алерты** 25](#_Toc198829610)

[**Глава 4.2: Централизованное логирование** 27](#_Toc198829611)

[**Глава 4.3: Планирование емкости и прогнозирование** 29](#_Toc198829612)

[**Глава 5: Политики управления рисками** 31](#_Toc198829613)

[**Глава 5.1: Политика доступа** 31](#_Toc198829614)

[**Глава 5.2: Chaos Engineering** 32](#_Toc198829615)

[**Глава 5.3: Антикризисные решения** 33](#_Toc198829616)

[**Глава 6: Эволюция архитектуры** 34](#_Toc198829617)

## **Введение**

Текущие темпы и объёмы разработки задают высокую планку производительности и надежности для современных IT систем. Разработанный нашей командой стандарт – поможет архитекторам и техлидам четко видеть преимущества от внедрения современных практик в существующую/разрабатываемую архитектуру

## **Глава 0: Надёжность физической и сетевой инфраструктуры**

Современные информационные системы строятся поверх инженерных и сетевых слоёв, надёжность которых критически важна для всех вышележащих компонентов: от k8s-кластеров до пользовательских приложений. Даже самая устойчивая программная архитектура теряет смысл, если в системе отсутствует электричество, сеть или базовые ресурсы ЦОД.  
Ниже описаны ключевые практики построения устойчивой инженерной и сетевой инфраструктуры, которые составляют фундамент общей модели надёжности.

## **Глава 0.1: Инженерная надежность**

Физическая устойчивость инфраструктуры начинается с надёжного ЦОД и его инженерных подсистем. Это базовый уровень пирамиды надёжности, на который опираются все остальные компоненты. К отказоустойчивым инженерным решениям можно отнести:

* **Электропитание с резервированием**: двойной ввод, ИБП, дизель-генераторы покрывающие потребности ЦОД по электропитанию
* **Системы охлаждения**: раздельное кондиционирование, прецизионные системы с резервированием, дублирование этих систем приветствуется.
* **Серверные стойки с резервом по климату и питанию**.
* **Физическая безопасность**: контроль доступа, видеонаблюдение, датчики среды.

Для систем 1-го тира рекомендуем размещение в ЦОД уровня Tier IV, с 2N резервированием всех инженерных систем. Для систем 2–3 тира — допускается Tier III с локальной гарантией электропитания. Системы 4-го тира допускают развёртывание в арендованных стойках или гиперскейлер-облаках с SLA ≥ 99.5%. Изображение выглядит как текст, диаграмма, снимок экрана, План

Контент, сгенерированный ИИ, может содержать ошибки.

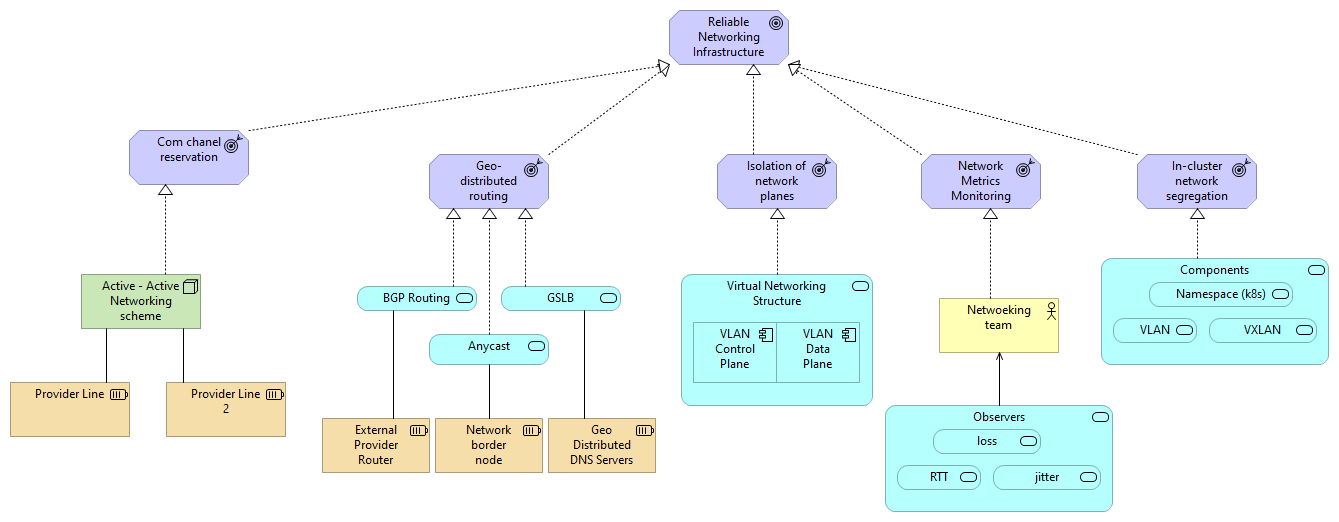
(Рис 0.1. Схема построения ресурсов ЦОД с разделением на тиры)

## **Глава 0.2: Сетевая архитектура**

Сетевая отказоустойчивость обеспечивает связность всех компонентов системы и возможность устойчивого маршрутизирования трафика даже при частичных сбоях. Потеря сетевого уровня — одна из самых трудно диагностируемых причин деградации систем, и должна быть исключена ещё на этапе проектирования.

Мы рекомендуем следующие практики:

* Резервирование каналов связи (**актив-актив** или **актив-пассив**, с разными провайдерами).
* Маршрутизация через **BGP** с приоритетами и fallback.
* Поддержка **Anycast или GSLB** для геораспределённых сервисов.
* Изоляция сетей уровня приложения и управления **(Data Plane / Control Plane)**.
* Мониторинг сетевых метрик в реальном времени **(loss, jitter, RTT)**.
* Разделение трафика внутри кластеров по **namespace/VLAN/VXLAN.**



(Рис 0.2. Схема построения надежной сетевой инфраструктуры)

Для систем 1–2 тира критично обеспечивать полную сетевую избыточность. Для 3–4 тира допускается откат на резервный канал связи и деградация производительности до SLA-порогов.

## **Глава 1: Культура разработки**

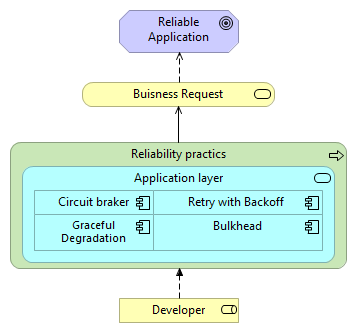
Одним из основополагающих факторов безопасной системы является непосредственно культура разработки программного обеспечения, можно создать совершенную архитектуру, которая будет учитывать все нюансы, но если отсутствует культура разработки, то все усилия по выстраиванию инфраструктуры могут пойти прахом, из-за критической уязвимости в ПО.

Далее указаны основные практики, которые мы рекомендуем внедрить для повышения надежности IT систем

## **Глава 1.1: Отказоустойчивая разработка**

Практики отказоустойчивой разработки позволяют облегчить работу DevOps-инженеров и архитекторов на этапе поддержки, создания и развития инфраструктуры. Применение таких подходов, как **Bulkhead**, **Circuit Breaker**, **Retry with Backoff** и **Graceful Degradation**, добавляет дополнительный слой защиты в информационные системы, позволяя приложениям продолжать работу даже при сбоях в инфраструктуре или внешних сервисах. Если что-то пойдёт не так на уровне аппаратного обеспечения, сетевых взаимодействий или сторонних API, программный уровень сможет компенсировать это за счёт заранее встроенных механизмов устойчивости. Ниже приводятся пояснения по каждой из этих практик:

* **Bulkhead** — изоляция компонентов системы друг от друга, чтобы сбой в одной части не распространялся на другие. Служит для того, чтобы один перегруженный модуль не положил все приложение.
* **Circuit Breaker** — механизм, прерывающий попытки обращения к сбойному сервису, если зафиксирована серия неудачных вызовов. Это позволяет избежать избыточной нагрузки и ускорить восстановление сервисов.
* **Retry with Backoff** — автоматическое повторение неудавшихся запросов с увеличивающимся интервалом между попытками. Это позволяет системе справляться с временными сбоями, не создавая избыточной нагрузки.
* **Graceful Degradation** — способность приложения снижать функциональность при возникновении сбоев, сохраняя при этом основную работоспособность. Например, при недоступности рекомендательной системы в интернет-магазине пользователь по-прежнему сможет оформить заказ. (очень важно на данном этапе синхронизироваться команде разработки с командой DevOps т.к это очень важный аспект)



(Рис 1.1. Схема построения процесса разработки с учетом практик надежности)

Для систем 1-го и 2-го тира следует применять представленные практики в полном объеме, для систем уровней 3 и 4 может применяться ограниченный набор

## **Глава 1.2: DevSecOps практики**

Процесс безопасной разработки с использованием практик DevSecOps очень важен для любой информационной системы с требованиями к надежности. Данный процесс позволяет выловить заложенные разработчиками уязвимости, ещё на этапе разработки/деплоя в dev среду. Благодаря полностью автоматизированным пайплайнам и выдаче результирующих отчетов – разработчики могут быть быстро проинформированы о найденных в их программном обеспечении уязвимостях – и устранить их перед выкатом в прод среду.

Мы предлагаем использовать схему с пайплайнам в системе контроля версий **GitLab** или аналоги (GitFlic, GitVerse) для непосредственной интеграции практик DevSecOps в процесс разработки. Также в стеке системы безопасной разработки присутствуют: open-source сервис для агрегации найденных уязвимостей **DefectDojo**, разнообразные сканеры кода для разных типов анализа:

* **SAST - Semgrep, KICS, Gitleaks** (можно дополнить другими сканерами отечественного производства например - PT Application Inspector)
* **SCA – DependencyTrack, Trivy –** (можно дополнить другими сканерами отечественного производства)
* **BCA – RAW: Kasperscky, KICS, maven – RND, npm – RND, pypi – RND, Images – Dependency Track.** (можно дополнить другими сканерами отечественного производства)
* **DAST – Owasp ZAP Browser, Owasp Zap Api, Nuclei -** (можно дополнить другими сканерами отечественного производства)

Между каждым из этапа проверки исходного кода или собранных артефактов есть так называемые **Quality Gates** – инструкции не позволяющие проверки продолжиться если на одном из этапов сканирования уязвимостей количество превысило ограничение на этап, или были найдены уязвимости уровней **Critical** и/или **High**

Изображение выглядит как текст, снимок экрана, диаграмма, линия

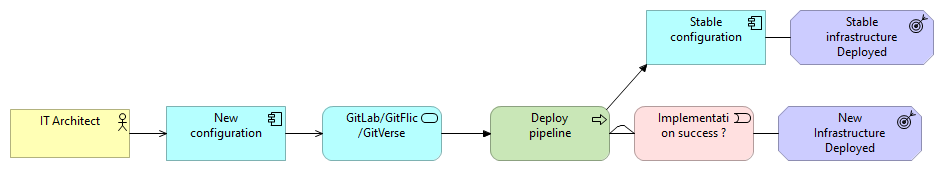
Контент, сгенерированный ИИ, может содержать ошибки.

(Рис 1.2. Схема процесса безопасной разработки)

Для систем 1-го и 2-го тира следует устанавливать порог уязвимостей Low и Medium как можно ниже, для систем тиров 3 и 4 могут допускаться чуть большие послабления или пропуск некоторых этапов тестирования

## **Глава 1.3: Безопасная инфраструктура (IaC)**

Подход IaС (infrastructure as code) широко применяется в крупных и современных информационных системах, потому что гарантирует единообразие развёртываемой архитектуры на неограниченном количестве машин, из-за того, что непосредственно архитектурные и инфраструктурные решения описываются с помощью кода в декларативном стиле с помощью таких инструментов как **Ansible** и **Terraform.** Применение данных инструментов способствует повышению надежности инфраструктуры практически исключая вклинение человеческого фактора на конфигурацию системы.

В рамках нашего стандарта мы рекомендуем заводить отдельный репозиторий для конфигураций Ansible и Terraform, дабы иметь версионированную историю изменений и имплементировать **Fallback** механизм в пайплайн развертки инфраструктуры по манифестам **Ansible** и **Terraform**. Данный механизм будет гарантировать что если система не заведется с новой версией манифестов, то будет использована последняя «стабильная» версия. 

(Рис 1.3. Схема процесса работы с версионированными конфигурациями инфраструктуры)

## **Глава 2: Архитектура кластера**

Высоконагруженные и отказоустойчивые ИТ-системы всё чаще строятся по кластерной модели. Кластер обеспечивает масштабируемость, гибкость и независимость вычислительных и сервисных компонентов. В рамках данной главы представлены практики и архитектурные подходы к построению надёжной кластерной инфраструктуры, включая уровни вычислений, сети, хранения и программного управления.

## **Глава 2.1: Топология и модели размещения**

Ключевым архитектурным решением нашего стандарта является **топология «2,5 ЦОД»** — подход, при котором инфраструктура развернута на трёх физических площадках:

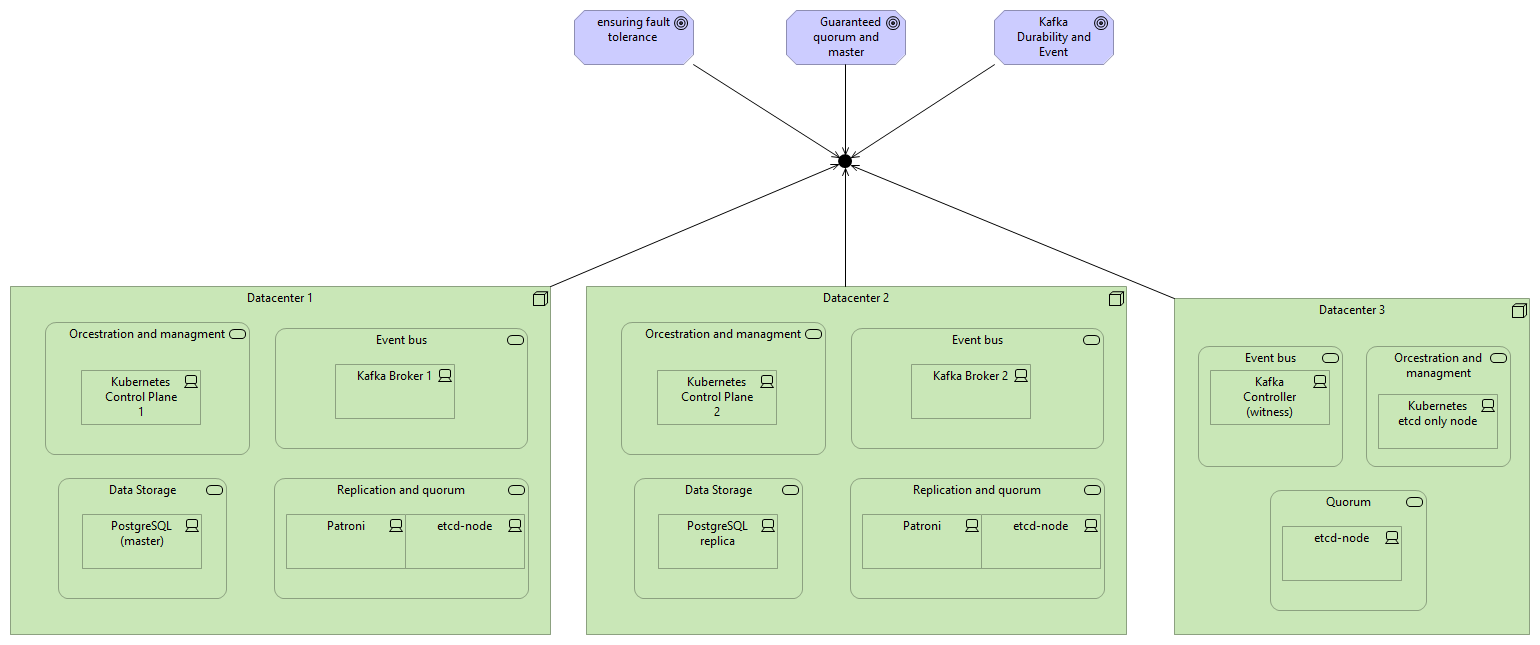
* Два полноценных дата-центра (**ЦОД-1** и **ЦОД-2**) с полными копиями сервисов и данных.
* Один облегчённый арбитражный узел (**ЦОД-3**) — используется только для обеспечения кворума и принятия решений в случае недоступности одного из основных ЦОД.

Этот подход обеспечивает:

* Высокую отказоустойчивость: система сохраняет работоспособность при сбое одного ЦОД.
* Поддержание консенсуса (кворума) при репликации баз данных, **Kafka**, **etcd** и других распределённых компонентов.
* Гибкость переключения между площадками без потери данных и нарушений **SLA**.

Реализация кворума с текущей топологией:

* Для кластеров **PostgreSQL** + **Patroni** + **etcd**:
  + 3 etcd-узла: по одному в каждом ЦОД.
  + Patroni отслеживает кворум через etcd и может переключить мастер на резервную площадку.
* Для **Kafka**:
  + Используется **min.insync.replicas** = 2, **ISR** (in-sync replicas) распределены по ЦОД-1 и ЦОД-2.
  + ЦОД-3 может держать контроллер **Kafka** (без партиций) для кворума.
* Для **Kubernetes**:
  + 3 **etcd**-ноды (ЦОД-1, ЦОД-2, ЦОД-3).
  + **Control Plane** узлы желательно держать в ЦОД-1 и ЦОД-2, ЦОД-3 — только **etcd**.



(Рис 2.1. Пример топологии отказоустойчивого кластера)

## **Глава 2.2: Сетевая интеграция и балансировка нагрузки**

Важной частью архитектуры кластеров является сеть. Нарушения в связности между компонентами, даже кратковременные, могут привести к каскадным отказам, недоступности сервисов и потере клиентского трафика. Надёжная архитектура должна предусматривать не только высокую пропускную способность, но и **избыточность, маршрутизацию, разделение потоков и сетевой контроль** на всех уровнях.

**1. Входной трафик (North-South, Ingress)**

Для обеспечения отказоустойчивого входа в систему мы рекомендуем использовать балансировщики нагрузки с поддержкой **health-check**, **TLS-терминации** и **перенаправления на рабочих ноды**:

* **F5 BIG-IP, HAProxy, NGINX-Ingress** — для L7 балансировки входящих HTTP/HTTPS-запросов.
* **MetalLB, Keepalived** — в bare-metal кластерах для обеспечения IP-failover.
* **Ingress-контроллеры с Canaries, A/B, Blue-Green** — как часть CI/CD пайплайнов.

Для систем 1-го тира следует использовать отказоустойчивый кластер L4/L7-балансировщиков с fallback-стратегией (например, F5 + резервный HAProxy).

**2. Внутренний трафик (East-West, Service Mesh)**

Микросервисная архитектура задает необходимость менеджмента вызовов между сервисами внутри кластера. Критические зоны, требующие контроля:

* **Трассировка запросов** (distributed tracing)
* **Политика доступа и авторизации**
* **Шифрование трафика**

Для реализации следует использовать:

* **Istio** — как полноценный Service Mesh с авторизацией, telemetry и circuit breaker'ами.
* **Consul Connect** — для интеграции с внешними системами.

В системах 2-го тира и выше рекомендуется использование Service Mesh с поддержкой L7 routing и TLS между сервисами.

**3. МежЦОДовое соединение и глобальная маршрутизация (DC-to-DC)**

Выбранная архитектура «2,5 ЦОД» требует отказоустойчивой связности между физическими площадками. Используем следующие практики:

* **Anycast** — для маршрутизации к ближайшему ЦОД по BGP. (подробнее в главе 0.2)
* **BGP with ECMP** — для отказоустойчивых многоканальных соединений между площадками. (подробнее в главе 0.2)
* **GSLB (Global Server Load Balancing)** — для распределения трафика между регионами или ЦОД. (подробнее в главе 0.2)
* **MPLS, GRE/IPSec** — для защищённого транспорта между регионами.

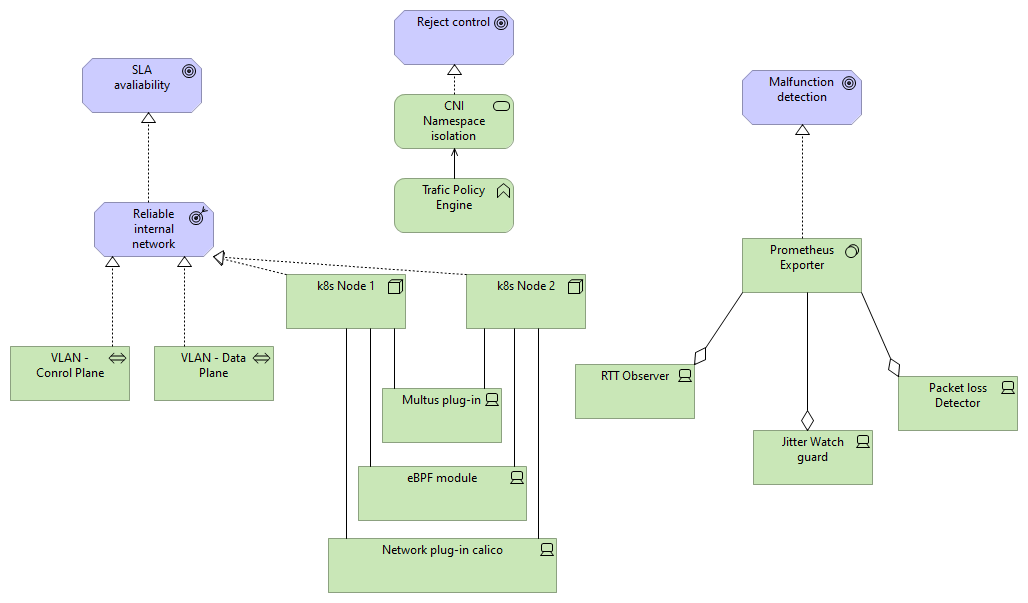
МежЦОДовая задержка должна быть ≤10 мс. Системы 1-го тира не должны допускать инцидентов типа **split-brain** ни в коем случае

**4. Сегментация сети: Control Plane / Data Plane**

Для предотвращения случайных или преднамеренных перегрузок Control Plane следует физически и логически изолировать его от клиентского и межсервисного трафика (подробнее в главе 0.2):

* **Multus** — для добавления нескольких интерфейсов в поды (разделение mgmt/data).
* **Calico + NetworkPolicy** — для контроля доступа и микросегментации.
* **Cilium** — для eBPF-базированной фильтрации и наблюдаемости на уровне ядра.

Все компоненты управления (**etcd**, **apiserver**, **ingress-controller**) должны находиться в Control Plane. Мониторинг, логирование и CI/CD — только через Data Plane.



(Рис 2.2. Пример построения надежной сетевой инфраструктуры на уровне кластера)

## **Глава 2.3: Оркестрация и жизненный цикл**

В рамках нашего стандарта базовым решением считается **K8s**, работающий в режиме **High Availability** с автоматическим восстановлением и масштабированием компонентов.

Оркестрация должна не только запускать контейнеры, но и обеспечивать **полный жизненный цикл приложения** — от инициализации и запуска до безопасного завершения и удаления, включая автоматическое масштабирование, диагностику, откат и соблюдение SLA стандартов, заданных критичностью сервиса.

**1. Жизненный цикл компонентов**

1. **Подготовка окружения**: загрузка образов, настройка конфигурации, подключение volume.
2. **Развёртывание**: через Deployment, StatefulSet, Job, DaemonSet.
3. **Масштабирование**: автоматическое или ручное.
4. **Отказ и восстановление**: реакция на сбои и перезапуск подов.
5. **Откат**: rollback при неудачном обновлении.
6. **Удаление**: завершение по lifecycle hook + finalizer.

**2. Механизмы устойчивости и контроля**

Для повышения надёжности рекомендуются следующие практики:

1. **Auto-healing**

* Kubernetes самостоятельно перезапускает упавшие контейнеры (restartPolicy).
* Использовать livenessProbe и readinessProbe для правильного отслеживания состояния.

1. **PodDisruptionBudget (PDB)**

* Ограничивает количество одновременно недоступных подов при плановых работах или обновлениях.

1. **Affinity/AntiAffinity**

* podAntiAffinity помогает избежать запуска одинаковых подов на одной ноде.
* Использовать topology-aware scheduling (zone, rack, hostname).

1. **Graceful Shutdown**

* Хуки preStop и terminationGracePeriodSeconds гарантируют точное время завершение работы сервиса.

**3. Масштабирование и ресурсная эффективность**

**Horizontal Pod Autoscaler (HPA)**

* Масштабирует количество реплик на основе метрик CPU, памяти или кастомных метрик (через Prometheus Adapter).

**Vertical Pod Autoscaler (VPA)**

* Автоматически увеличивает/уменьшает ресурсы requests и limits у подов.

**Cluster Autoscaler**

* Добавляет/удаляет worker-ноды на основе загрузки кластера.
* Настроен с предсказанием пиковой нагрузки (по метрикам за прошлые сезоны).

**4 Управления конфигурациями и секретами**

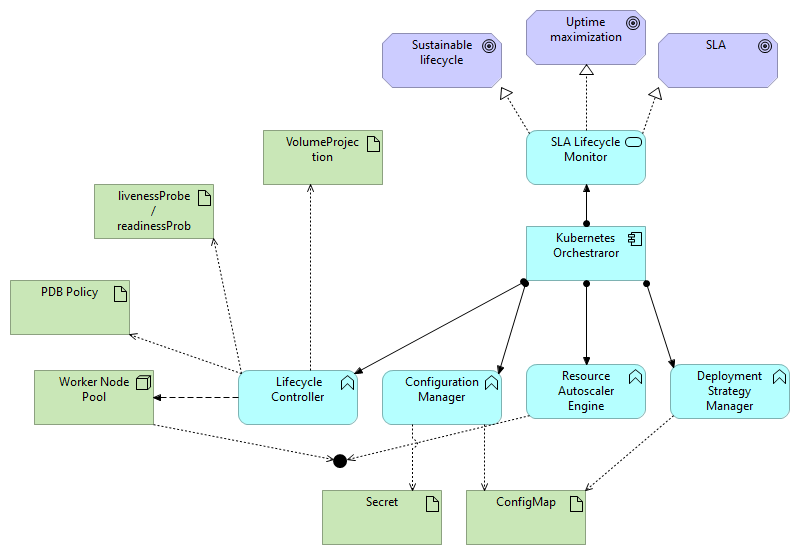
* Все переменные окружения должны храниться в ConfigMap и Secret в k8s
* Использовать immutable config maps
* Использовать VolumeProjection для atomic mounts

**5 Автоматическое устранения последствием сбоя**

В рамках системы рекомендуется имплементировать механизмы автоматического устранения последствий сбоев.

* **livenessProbe + restartPolicy –** инструкции на автоматический рестарт зависших/вылетевших компонентов.
* **Внутренние Health-Check –** помогают своевременно распознать сбой
* **VolumeAttachment recovery –** при потере pvc запускается механизм восстановления

Гарантировать полностью автоматическое восстановление системы довольно трудно из-за огромного количества факторов, поэтому помимо систем восстановления также важно озаботиться системой информирования, например через **Prometheus Alert**



(Рис 2.3. Пример построения схемы оркестрации и жизненного цикла)

## **Глава 2.4: Управление обновлениями**

Основной целью данной главы является минимизация влияния частых релизов (2 раза в неделю вторник и суббота) на пользователей. Также наш стандарт гарантирует возможность быстрого rollbackА на предыдущую стабильную версию.

**1. Стратегии выкатки новых версий**

* **Blue/Green Deployment –** одновременно работают обе версии сервиса, переключение трафика происходит автоматически через ingress. Гарантирует мгновенный откат на предыдущую версию. Реокмендуем использовать этот подход для сервисов с 1-ым и 2-ым тиром критичности
* **Canary Deployment –** поэтапное обновление «по нарастающей» - начинаем с 5% трафика и увеличиваем вплоть до 100. Позволяет выявить проблемы на раннем этапе. Рекомендуем использовать для сервисов 2-го и 3-го тира
* **Feature Toggles –** включаем новую функциональность через конфигурационные флаги – основная логика разворачивается, но остается неактивной.

**2. Инструменты и механизмы**

* **Argo Rollouts –**  Управление Canary и Blue/Green через CRD**Canary**
* **Flagger –** Автоматизация Canary на базе метрик.
* **GitOps -** Контроль версии и истории изменений (интегрируется с процессом безопасной разработки DevSecOps)
* **Helm -** Шаблонизация манифестов и rollback
* **Skaffold/Kustomize -** Упрощение сборки и доставки

**3. Метрики и мониторинг обновлений**

Рекомендуем отслеживать следующие метркии:

* **Error Rate / HTTP 5xx**
* **Response Time / Latency**
* **Pod CrashLoopBackOff / Restart Count**
* **Log patterns**
* **Нагрузка в определенные периоды (**для выделения доп ресурсов при выкатке новых версий в пиковые периоды**)**

**4. Rollback / откат**

Требования к реализации Rollback механизма – Максимальная автоматизация (Любое человеческое вмешательство неизбежно приведет к задержкам), Безопасность, Контролируемость. Рекомендуем использовать:

* **Kubectl rollout undo**
* **Helm rollback**

**5. Рекомендации**:

* **Проведение нагрузочного тестирования перед запуском CI/CD пайплайна –** мы рекомендуем провести нагрузочное тестирование с учетом метрик по возрастающей нагрузке в периоды интенсивного использования сервиса дабы не возникло проблем в будущем

Изображение выглядит как текст, Самоклеющийся листок, диаграмма, План

Контент, сгенерированный ИИ, может содержать ошибки.

(Рис 2.4. Пример схемы обновления микросервисов)

## **Глава 3: Хранение и передача данных**

Система хранения данных ключевой компонент информационной системы. Наш стандарт задает четкие методы и правила для сбора, хранения репликации и синхронизации данных между ЦОД.

## **Глава 3.1: Уровни хранения**

С ростом объема данных необходимость в их разделении по частоте доступа становится критической. Мы предлагаем внедрить многоуровневую модель хранения, основанную на принципах hot/cold хранения и data lifecycle management.

**Классификация данных:**

* **Hot data (оперативные данные)** — часто используемые, имеют критичное значение. Примеры: заказы, активные пользователи, сессии.
* **Warm data (реже используемые)** — исторические транзакции, выгрузки для BI-систем.
* **Cold data (архивы)** — данные, к которым редко обращаются (например, отчёты 5-летней давности), но которые обязаны храниться по регламенту организации.
* **Log data** — технические и аудит-логи, хранятся ограниченное время (по умолчанию 30 дней).

**Технологические рекомендации:**

* **Hot**: PostgreSQL с автошардированием или Redis/Tarantool с персистентностью.
* **Warm**: PostgreSQL partitioned tables.
* **Cold**: объектное хранилище (MinIO/S3/Glacier), возможно со сжатием (zstd).
* **Log**: ELK/Loki/Graylog + ротация логов по TTL.

**Экономическая целесообразность:**

* **Горячее хранение на SSD / оперативной памяти** – требует наибольших затрат. RAM-based СУБД стоят кратно дороже холодных хранилищ на HDD, но каждая секунда задержки при использовании HDD – стоит бизнесу денег (например торговля на бирже или другие фин операции). Следует использовать только для критически важных систем
* **Тёплое хранение** – сбалансированное решение по соотношению цена/доступность используется в связке с PostgreSQL partitioned или другими СУБД подойдет для тира 2 и 3. В пересчете на рубли/объем теплое хранилище будет на 300–400% эффективней горячего
* **Холодное хранение** – объектные хранилища на HDD. Наименьшая скорость – наибольший объём
* **Log** – В хранении логов самое главное – ротация, она поможет сэкономить огромное количество дискового пространства. Логи не хранятся долго поэтому настроив политику ротации можно сильно сэкономить.

**Управление сроками хранения:**

* Операционные данные — до 3 лет в онлайне.
* Архивные данные — до 5 лет в cold-хранилищах.
* Логи — 30 дней (автоматическая ротация).

Для пермещения данных используем специальный DataLifecycle manager

Для систем 1–2 тира используется только SSD и реплицируемая база; cold хранение отложено через CDC/Debezium.  
Для 3–4 тира допускается использование HDD и менее затратного хранения без онлайнового индекса.

Изображение выглядит как текст, снимок экрана, Самоклеющийся листок, диаграмма

Контент, сгенерированный ИИ, может содержать ошибки.

(Рис 3.1. Пример схемы разделения уровней хранения)

## **Глава 3.2: Репликация и кворум**

Применение реплицируемых СУБД и брокеров сообщений позволит минимизировать потери при сбоях и ускорить восстановление после них.

**Основные модели репликации:**

* **Primary/Secondary (Master/Replica)** — базовая модель с одним узлом записи (master), остальные только чтение (replica).
* **Multi-master** — Несколько узлов с поддержкой записи, синхронизация между ними. (напр. Tarantool или PostgreSQL-BDR).
* **Кворумная репликация** — Подтверждение записи при достижении кворума (Kafka, etcd, Patroni).

**Инструменты:**

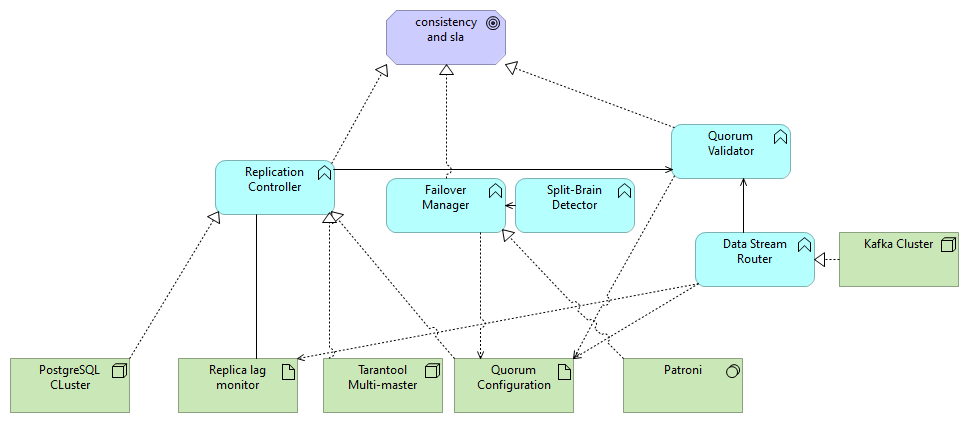
* PostgreSQL + Patroni + etcd — для отказоустойчивой синхронной репликации.
* Kafka с ISR (in-sync replicas) и контролем доставки.
* Tarantool с hot standby и failover.

**Рекомендации:**

* Обязательное логирование WAL (Write-Ahead Logging) и сохранение на независимых физических дисках.
* Контроль задержек реплик (replica lag) и механизмов переключения лидеров.
* Интеграция процесса разрешения конфликта split-brain (за счет кворума).

Для систем 1-го и 2-го тира — только синхронная репликация с автоматическим failover.

Для Тиров 3 и 4 возможна асинхронная репликация с деградацией доступности при сбое.



(Рис 3.2. Пример схемы репликации данных)

## **Глава 3.3: Интеграция и согласованность данных**

По топологии “2.5 ЦОД” Дата-центры взаимодействуют между собой через API, брокеры сообщений, шины данных, и при этом необходимо обеспечить как доставку, так и согласованность информации.

**Модели согласованности:**

* **Strong Consistency** — все участники всегда видят одно и то же состояние постоянно
* **Eventual Consistency** — временное расхождение в данных которое исправляется со временем
* **Causal Consistency** — упорядоченная доставка событий

**Механизмы:**

* **Саги (Saga Pattern)** — каскадные транзакции с возможностью компенсации.
* **Idempotent operations** — защита от повторной доставки.
* **Outbox Pattern** — интеграция базы и брокера сообщений.
* **Retry / Dead-letter queue** — повтор доставки и контроль ошибок.

**Инструменты:**

* Apache Kafka, RabbitMQ, NATS — для сообщений.
* Debezium — для CDC (change data capture).
* API Gateway + Circuit Breaker — защита интеграционных вызовов.

Для систем тира 1 интеграций предпочтительна Strong или Causal Consistency.

Для тира 2–3–4 допустимо Eventual с ретрансляцией ошибок и alert-уведомлениями. Изображение выглядит как текст, диаграмма, снимок экрана, Самоклеющийся листок

Контент, сгенерированный ИИ, может содержать ошибки.

(Рис 3.2. Пример схемы согласования данных)

## **Глава 4: Мониторинг и управление инцидентами**

Cистема мониторинга и отлаженный процесс управления инцидентами. без своевременного обнаружения и корректной обработки происшествий все усилия по обеспечению надёжности могут оказаться тщетными: сбои остаются незамеченными, реакция затягивается, ущерб растёт, а доверие пользователей и бизнеса снижается.  
Ниже мы привели основные практики и инструменты, которые мы рекомендуем внедрить для эффективного мониторинга и быстрого реагирования на инциденты.

## **Глава 4.1: Метрики и алерты**

Эффективный мониторинг основан на двух взаимосвязанных элементах: сборе ключевых показателей (метрик) и своевременном срабатывании алертов. Метрики дают количественную картину здоровья приложения и инфраструктуры, а алерты — автоматические уведомления при выходе показателей за заранее определённые границы. Без чёткого определения того, что и как мы измеряем, а также без механизмов оперативного оповещения, риски незамеченных инцидентов и длительных простоев существенно возрастают.

Изображение выглядит как текст, Самоклеющийся листок, диаграмма, План

Контент, сгенерированный ИИ, может содержать ошибки.

(Рис 4.1. Пример схемы сбора метрик и отправки алертов)

**Сбор и агрегация метрик**

* Объект InfraMetricsCollector собирает метрики с уровня инфраструктуры (CPU, RAM, disk io)
* App metrics collector – собирает метрики с уровня приложений (RPS, ошибки и т.д)
* Оба компонента далее передают данные в MetricIngestionService – промежуточный слой, который обхединяет все собранные метрики в унифицированный формат.
* Aggregation Service – производит группировку и downsampling

**2. Визуализация и отчётность**

* Dashboard Service - отображает ключевые показатели (Grafana)
* Reporting Service – формирует отчеты по SLA, SLO, трендам и отклонениям

**3. Генерация алертов и управление инцидентами**

Для начала следует установить пороговые значения для срабатывания алерта:

* Alert Defenition – набор правил (к примеру CPU > 80%)
* Alert Rule Engine – применение правил к собранным метрикам
* Notification API – сервис через который алерт уходит в систему уведомлений
* Notification dispatcher – маршрутизирует запросы в разные системы уведомления

## **Глава 4.2: Централизованное логирование**

Каждый компонент генерирует собственные журналы событий. Чтобы оперативно обнаруживать и устранять сбои и проводить сквозной анализ поведения приложений, логи собирают и обрабатывают централизованно в составе единого Observability‑стека (логи + метрики + трейсинг).

**Основные цели и преимущества**

1. **Удобство поиска и анализа**  
   – Полнотекстовый поиск по всем узлам и сервисам.  
   – Интерактивные дашборды и агрегированные отчёты по логам.
2. **Соответствие регламентам**  
   – Стандартизованный аудит и отчётность (GDPR, PCI DSS).  
   – Маскирование PII и гибкая система прав доступа.

**Лучшие практики**

1. **Структурированный и семантический лог**  
   – Каждая запись в виде JSON‑объекта с конкретными и предсказуемыми полями (унифицирование лога, как объекта)  
   timestamp, level, service, trace\_id, message, context.
2. **Корреляция запросов**  
   – Передача trace\_id/request\_id через весь вызов‑цепочку для объединения событий в единую сессию.
3. **Гибкая маршрутизация**  
   – Отдельные каналы для «сырых» (raw) и «обогащённых» (enriched) логов с добавлением метаданных: геолокация, пользовательские атрибуты, окружение.
4. **Регулируемые уровни шума**  
   – Чёткая политика уровней логирования (ERROR / WARN / INFO / DEBUG).  
   – Возможность временно повышать детализацию (on‑demand debugging) без перезагрузки сервисов.
5. **Ротация и управление жизненным циклом**  
   – Настройка Index Lifecycle Management (ILM):  
   • «горячее» хранение — 7 дней,  
   • «тёплое» — 30 дней,  
   • архив в S3‑совместимом хранилище.
6. **Алертинг по логам**  
   – Правила на ключевые паттерны и частотные аномалии (например,> 100 ошибок/минуту).

– Алерты не только по числовым метрикам, но и по текстовым паттернам (например, резкий рост ошибок “500 Internal Server Error”).

**Безопасность и целостность логов**

1. **Шифрование канала и на диске**  
   TLS/mTLS между агентами и брокерами, SSL/TLS для Elasticsearch, шифрование данных «в покое» и «в пути».
2. **Аутентификация и авторизация**  
   Использование SASL/SCRAM (Kafka), RBAC (Elasticsearch, Graylog), ограничение прав агентов и пользователей по принципу «минимально необходимого».
3. **Журнал неизменяемости (WORM / append‑only)**  
   Настройка и хранение логов так, чтобы записи нельзя было модифицировать или удалять до окончания периода хранения.
4. **Подпись и контроль целостности**  
   Добавление цифровых подписей или HMAC‑хэша к блокам логов, чтобы быстро обнаружить попытки фальсификации.

Централизованная система логирования — основа для быстрой диагностики и восстановления. Она предоставляет единый источник правды для root‑cause анализа, позволяет безопасно воспроизводить инциденты в тестовом окружении и поставляет данные в инструменты алертинга и оркестрации. Для автоматического масштабирования или перезапуска упавших контейнеров используются отдельные механизмы (Kubernetes HPA, health‑checks), но логи играют ключевую роль в триггерах этих процессов и в построении AIOps‑решений для прогнозирования и предотвращения отказов.

## **Глава 4.3: Планирование емкости и прогнозирование**

Метрики и алерты (§ 4.1) дают представление о текущем состоянии сервисов, а централизованное логирование (§ 4.2) — полный контекст событий. Планирование емкости и прогнозирование нагрузки дополняют эти механизмы проактивным подходом: на основании исторических данных и трендов заранее рассчитываются требования к ресурсам, что позволяет избежать простоя и оптимизировать затраты.

1.  **Цели и задачи**

* Гарантировать соответствие SLA/SLO при пиковых и географически распределённых нагрузках.
* Определять «горячие» сервисы, кластеры и регионы до появления инцидентов, а не отдельные узлы.
* Централизованно управлять ресурсным бюджетом (FinOps) и оптимизировать затраты.
* Включить capacity planning в релизный цикл и процесс управления изменениями.

2.  **Этапы процесса**

* **Агрегация данных**: сбор исторических SLI‑метрик и логов по namespace/label‑группам в Observability‑стеке.
* **Анализ трендов**: построение P95‑P99 latency, p95 CPU/memory, анализ дневной, недельной, сезонной сезонности и кампаний.
* **Сценарное моделирование**: стресс‑тесты и нагрузочные проверки в staging‑кластере с production‑конфигурацией.
* **Прогнозирование**: применение ARIMA/SARIMA и ML‑алгоритмов (Prophet, XGBoost) для оценки будущего потребления.
* **Формирование плана**: стратегии горизонтального (дополнительные инстансы, шардирование) и вертикального масштабирования, multi‑AZ/region резервирование.
* **Валидация**: Canary‑тесты и ежеквартальные Capacity Review Board для корректировки допущений.

3.  **Лучшие практики**

* **Единый источник правды**: SLI/SLO, метрики и логи хранятся в одном стекe (Prometheus + Grafana).
* **Автоматизация**: интеграция с Kubernetes Cluster Autoscaler, облачными predictive autoscaler’ами.
* **Headroom резерв**: 20–30 % сверх прогноза на уровне кластера и подкластеров (может варьироватся от тира системы).
* **Capacity Review Board**: регулярные встречи Dev, Ops и FinOps для анализа отклонений и бюджета ресурсов.
* **Гибридный подход**: комбинировать классические статистические модели, ML‑прогнозы и экспертную донастройку.

4.  **Инструменты и методы**

* **Observability**: Prometheus, Grafanа.
* **: GCP** Vertex AI, open‑source ARIMA/SARIMA библиотеки, Любые ml-практики.
* **Нагрузочное тестирование**: Gatling, Locust в распределённом исполнении.
* **BI и PlanOps**: Tableau, Power BI, (любые другие инструменты визуализации Data Lens).

Таким образом, продуманное ёмкостное планирование на уровне сервисов и кластеров позволяет масштабироваться в периоды повышенной нагрузки, опираясь на исторические данные и прогнозирование, минимизировать риски деградации и держать под контролем облачные и аппаратные расходы.

## **Глава 5: Политики управления рисками**

В этом разделе мы поговорим о необходимых политиках в компании для поддержания общей надежности системы и предотвращении потенциальных антропогенных факторов, влияющих на SLA

## **Глава 5.1: Политика доступа**

Важная система крупной компании может оказаться лакомой целью для хакеров, поэтому важно изолировать систему от нежелательного влияния не только извне на стороне юзеров, но и на более продвинутом уровне.

1. Использовать **VPN** (virtual private tunel) для доступа к серверам с развернутой инфраструктурой – данная мера предосторожности поможет изолировать доступ к серверам по ssh извне. Сертификация и выдача конфигураций vpn протоколов для подключения должна быть реализована с помощью отдела безопасности в компании.

2. Регулярное **Pentest-тестирование** на проникновение – регулярный аудит инфраструктуры поможет выявить неявные «дыры» в инфраструктуре и залатать их до наступления катастрофических последствий

3. **BugBounty** – реализация программ BugBounty для независимых pentest и security инженеров. Объявив награду за нахождения критических багов в вашей системе и программном обеспечении, вы можете сильно сэкономить в будущем на восстановлении инфраструктуры или издержках проникновения

Изображение выглядит как текст, Самоклеющийся листок, диаграмма, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

(Рис 5.1 Триада политики доступа и безопасности)

## **Глава 5.2: Chaos Engineering**

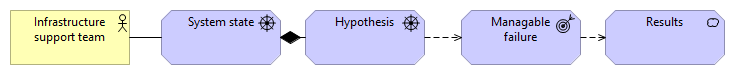
Как говорится: «Тяжело в учении – легко в бою», применяя практики Chaos Engineering можно подготовить команду к критическим ситуациям и непредвиденным эксцессам, чтобы время их реакции и понимание происходящего были на высоких показателях.

Рассмотрим плюсы данной методологии. Команда поддержки сможет изучить как система реагирует на сбои, где находятся слабые места и как их можно усилить, работает ли автоматика восстановления от переключения на резервный ЦОД до пера развёртывания сервиса.

**Принципы Chaos Engineering:**

* **Определить** - какое состояние для системы является нормальным (RPS, процент ошибок и прочее)
* **Подобрать гипотезу** - «Если отстрелит ЭТО, то ТО должно продолжать работать»
* **Произвести контролируемый сбой** – отключить узел или перегрузить сеть
* **Наблюдать –** как изменились метки, какие механизмы сработали и прочее

В качестве основного инструмента для Chaos Engineering мы рекомендуем использовать инструмент Chaos Mech для k8s позволяющий выставлять таймауты, убивать сущности и совершать IO-fault.



(Рис 5.2 Схема проведения CE)

## **Глава 5.3: Антикризисные решения**

В отличии от проведения мероприятий по типу Chaos Engineering. Данная глава отвечает за непосредственное планирование критической ситуации включающие ответы на такие вопросы как “Кто отвечает за принятие решений в случае ЧП?”, “Кто является актором в случае чрезвычайной ситуации?”, “Кто должен быть проинформирован?” и тд и тп.

Также стоит разработать конкретные документы **Failover Plan** c документацией на случай ЧП, с информацией о том на какой ЦОД идет переподключение, Какие сервисы переключатся автоматически, а какие требуют ручного переключения, Необходимые скрипты и прочее.

Все эти решения должны быть имплементированы в максимально доступном формате и продублированы несколько раз. Нашей рекомендацией является следующая схема: настроить Prometheus Alert на рассылку антикризисных инструкций ответственным людям, сохранить все инструкции в Markdown или любом другом формате на независимом хранилище и наконец продублировать все материалы в печатном виде.

## **Глава 6: Эволюция архитектуры**

**Эволюция архитектуры как управляемый процесс**

Развитие архитектуры должно проходить на основе:

* обратной связи от эксплуатации (метрики, инциденты),
* технологических трендов и появляющихся рисков
* результатов, проведённых CE
* стратегических бизнес-целей (масштабирование, новые рынки, интеграции).

В рамках стандарта надёжности предлагается внедрить **регулярные архитектурные обзоры (architecture review board)**, на которых обсуждаются:

* предложенные улучшения,
* выявленные технические долги,
* миграции устаревших решений,
* экспериментальные подходы и Proof of concept

**Заключение**

Надёжность ИТ-систем — это **не конечная цель**, а **непрерывный процесс**, в котором участвуют все: архитекторы, разработчики, DevOps-инженеры, службы поддержки и бизнес.

Проектируя инфраструктуру с учётом отказоустойчивости, управляя рисками, контролируя конфигурации и тестируя предельные сценарии, мы создаём систему, способную адаптироваться, развиваться и выдерживать внешние воздействия.

Данный стандарт является базовым сводом правил для построения надежной и отказоустойчивой информационной системы в современных реалиях, его можно спокойной развивать и упрощать некоторые аспекты подгоняя под требования и реалии проекта