ТЕХНИЧЕСКОЕ РУКОВОДСТВО • ЧАСТЬ 1 (ЯДРО КНИГИ)

АРХИТЕКТУРА И ПРАКТИКА KATA CONTAINERS (2025–2026)

Полное техническое руководство: от низкоуровневого устройства `runtime-rs` и гипервизора Dragonball до изоляции AI-агентов и Confidential GPU

Глава 1. Проблема изоляции в современных облачных средах и предпосылки создания Kata Containers

1.1. Анатомия стандартных контейнеров (`runc`) и вектор атак на общее ядро

Традиционная контейнеризация, ставшая стандартом де-факто благодаря Docker и спецификации OCI (Open Container Initiative), опирается на стандартный рантайм `runc`. С точки зрения операционной системы Linux, контейнер не является объектом первого класса — это группа обычных процессов на хосте, изолированных с помощью трех базовых механизмов ядра:

1. **Пространства имен (Namespaces):** ограничивают видимость ресурсов (PID, Network, Mount, IPC, UTS, User, Cgroup). Процесс внутри контейнера считает, что имеет `PID 1`, но в глобальной таблице процессов ядра хоста он фигурирует под обычным идентификатором (например, `PID 48219`).
2. **Контрольные группы (cgroups v1/v2):** отвечают за учет и лимитирование потребления аппаратных ресурсов (квоты CPU, лимиты оперативной памяти, IOPS блочных устройств, сетевая пропускная способность).
3. **Мандатное управление доступом и фильтрация системных вызовов (Seccomp-BPF, AppArmor, SELinux, Linux Capabilities):** урезают права root-пользователя внутри контейнера и блокируют опасные системные вызовы (например, `kexec_load`, `init_module`, `mount`).

Фундаментальная проблема безопасности: все контейнеры на одном физическом или виртуальном узле разделяют **единое ядро Linux хоста**. Системный интерфейс ядра Linux насчитывает более 350–450 системных вызовов (`syscalls`). Даже при включенном профиле `seccomp` по умолчанию в Docker/containerd приложению внутри контейнера остаются доступными около 300 системных вызовов. Любая уязвимость в коде обработчиков этих вызовов, файловых систем, сетевого стека или подсистемы `io_uring` внутри ядра хоста приводит к побегу из контейнера (Container Escape).

РЕАЛЬНЫЕ АРХИТЕКТУРНЫЕ УЯЗВИМОСТИ ПОСЛЕДНИХ ЛЕТ, ДЕМОНСТРИРУЮЩИЕ НЕДОСТАТОЧНОСТЬ RUNC ДЛЯ ЗАПУСКА НЕДОВЕРЕННОГО КОДА (UNTRUSTED WORKLOADS)

- **CVE-2022-0185 (Heap-based buffer overflow в `legacy_parse_param`):** ошибка в подсистеме Filesystem Context ядра Linux позволяла процессу с правами `CAP_SYS_ADMIN` внутри собственного пользовательского пространства имен (User Namespace) перезаписать память ядра и получить полные привилегии на хосте.
- **CVE-2022-0847 (Dirty Pipe):** уязвимость в механизме инициализации флагов буферов каналов (`pipe_buffer`), позволявшая непривилегированному процессу перезаписывать страницы памяти в кэше страниц (page cache) для файлов, открытых только на чтение, включая сам бинарный файл `/proc/self/exe` рантайма `runc` на хосте.
- **CVE-2024-21626 (Leaky Vessels в `runc`):** утечка файлового дескриптора рабочей директории хоста внутрь контейнера при некорректной обработке `/proc/self/fd/`, что позволяло злоумышленнику получить доступ к корневой файловой системе узла.
- **Уязвимости в `io_uring`:** асинхронный интерфейс ввода-вывода `io_uring` оказался настолько сложным и подверженным ошибкам работы с памятью (Use-After-Free, Race Conditions), что инженеры Google были вынуждены полностью отключить его в продакшене для многих публичных сервисов, так как свыше 60% эксплойтов, присылаемых в программу Bug Bounty по ядру Linux, эксплуатировали именно `io_uring`.

1.2. Подходы к песочницам (Sandboxing): Перехват syscalls (gVisor) против аппаратной виртуализации (Kata)

Для решения проблемы общего ядра в индустрии сформировались два принципиально разных архитектурных подхода:

Подход А: Перехват системных вызовов и ядро в пространстве пользователя (User-space Kernel — Google gVisor)

Рантайм `runsc` (gVisor) внедряет между контейнером и ядром хоста прослойку под названием **Sentry** (написана на Go). Sentry выступает в роли гостевого ядра, работающего в непривилегированном пространстве пользователя (Ring 3). Когда процесс внутри контейнера выполняет системный вызов (например, `open()`, `socket()`, `read()`), он перехватывается механизмом `systrap` (ранее `ptrace` или `kvm`) и обрабатывается кодом Sentry, который самостоятельно реализует логику виртуальной памяти, TCP/IP-стек (netstack) и файловую систему, обращаясь к реальному ядру хоста только через строго ограниченный набор из ~50 безопасных системных вызовов.

- **Плюсы gVisor:** минимальный оверхед по памяти на старте, быстрое время инициализации, не требует обязательной поддержки вложенной аппаратной виртуализации (Nested Virtualization).
- **Минусы gVisor:** 1) *Неполная совместимость по системным вызовам:* сложные или специфичные системные вызовы (`ioctl` для специфического железа, прямая работа с GPU, сложные флаги `mmap` или `bpf`) не поддерживаются или работают иначе, чем в нативном Linux. 2) *Просадка производительности на системных вызовах и I/O:* каждый системный вызов требует переключения контекста и программной эмуляции. Приложения с высокой интенсивностью файлового ввода-вывода (БД, сборщики компиляторов, обработка мелких файлов) или сетевого трафика теряют от 20% до 60% производительности по сравнению с нативным запуском.

Подход Б: Аппаратная изоляция на базе легковесных виртуальных машин (Lightweight MicroVMs — Kata Containers)

Kata Containers помещает каждый контейнер (или под целиком в терминологии Kubernetes) внутрь отдельной легковесной виртуальной машины, изолированной на уровне процессора с помощью расширений аппаратной виртуализации (**Intel VT-x / AMD-V**, **ARM Virtualization Extensions**). Внутри каждой микро-ВМ запускается собственное, минимально сконфигурированное **гостевое ядро Linux** (Guest Kernel).

- **Плюсы Kata Containers:** 1) *100% совместимость с Linux ABI:* поскольку внутри микро-ВМ работает настоящее ядро Linux, любое приложение, системная утилита, драйвер или бинарный файл работает без каких-либо модификаций. 2) *Нативная скорость выполнения вычислений и системных вызовов:* системные вызовы обрабатываются гостевым ядром напрямую на аппаратном уровне (в режиме VMX non-root operation) без участия ядра хоста и без дорогих выходов в гипервизор (VM Exits). 3) *Аппаратный барьер безопасности:* даже если злоумышленник внутри контейнера проэксплуатирует zero-day уязвимость ядра Linux (например, в том же `io_uring`), он получит права root только внутри своей эфемерной микро-ВМ, где нет данных других клиентов и нет доступа к физическому хосту. 4) *Поддержка аппаратного проброса устройств (VFIO):* возможность напрямую пробросить в контейнер физическую сетевую карту (SR-IOV) или графический ускоритель (GPU) для задач машинного обучения.

1.3. Сводная сравнительная таблица архитектур изоляции

Критерий	Стандартный `runc`	Google `gVisor` (`runsc`)	`Kata 3.x` (Go + CHV)	`Kata 4.0` (`runtime-rs` + Dragonball)
Граница безопасности	Программная (Namespaces, Cgroups, Seccomp)	Программная (User-space ядро Sentry на Go)	Аппаратная (Intel VT-x / AMD-V, KVM)	Аппаратная (Intel VT-x / AMD-V, KVM) + Memory Safety (Rust)
Ядро контейнера	Общее ядро хоста	Эмулируемое ядро Sentry	Выделенное гостевое ядро Linux	Выделенное гостевое ядро Linux
Совместимость с ABI	100%	~85–90% (ограничения по	100% (нативное ядро Linux)	100% (нативное ядро Linux)

Глава 2. Эволюция архитектуры: от Kata 1.x к Kata 4.0 (2025–2026)

Чтобы понимать, как устроены современные версии Kata Containers, необходимо проследить, как инженеры избавлялись от лишних уровней абстракции и межпроцессного взаимодействия (IPC).

2.1. Первое поколение (Kata 1.x): Многопроцессная схема и тяжелый IPC

Когда в конце 2017 года проекты Intel Clear Containers и Hyper runV объединились под эгидой OpenStack Foundation (ныне OpenInfra Foundation), архитектура Kata 1.x пыталась мимикрировать под интерфейс классического `runc`. Для запуска **одного** контейнера или пода на хосте порождалась целая цепочка из **пяти независимых процессов**:

1. `containerd-shim`: стандартный шим от `containerd`.
2. `kata-runtime`: CLI-утилита, парсившая OCI-спецификацию.
3. `kata-shim`: прокси-процесс для перехвата потоков ввода-вывода (`stdin` / `stdout` / `stderr`) контейнера.
4. `kata-proxy`: мультиплексор соединений, переводивший gRPC-запросы с хоста в последовательный порт или `vsock` виртуальной машины.
5. `qemu-system-x86_64`: сам процесс гипервизора.

В качестве файловой системы использовался устаревший протокол **`virtio-9p`** (наследие ОС Plan 9), который не поддерживал кэширование страниц гостевой ОС должным образом и превращал сборку кода или работу с базами данных в крайне медленный процесс. На один пустой контейнер тратилось от **80 до 130 МБ оперативной памяти** только на обслуживание управляющих процессов на хосте.

2.2. Второе и третье поколение (Kata 2.x – 3.x): Переход на **`Shim v2`** и **`virtio-fs`**

С выходом интерфейса **Containerd Runtime v2 (Shim v2 API)** в Kata 2.0 произошла первая крупная архитектурная чистка:

- Процессы `kata-runtime`, `kata-shim` и `kata-proxy` были объединены в единый долгоживущий процесс на под — **`containerd-shim-kata-v2`** (написанный на Go).
- Протокол связи между хостом и `kata-agent` внутри микро-ВМ был переведен с тяжелого gRPC (поверх HTTP/2) на легковесный бинарный протокол **`ttRPC`** (ttrpc — версия gRPC, специально оптимизированная для сред с малым объемом памяти, работающая напрямую поверх потоковых сокетов `AF_VSOCK` без накладных расходов HTTP/2).
- Протокол файловой системы `virtio-9p` был заменен на **`virtio-fs`** (разработанный инженерами Red Hat на базе протокола FUSE поверх очередей `virtio`). Вместе с механизмом **DAX (Direct Access)** это позволило отображать кэш страниц хоста напрямую в адресное пространство гостевой ВМ через окно памяти PCI BAR, минуя копирование данных.

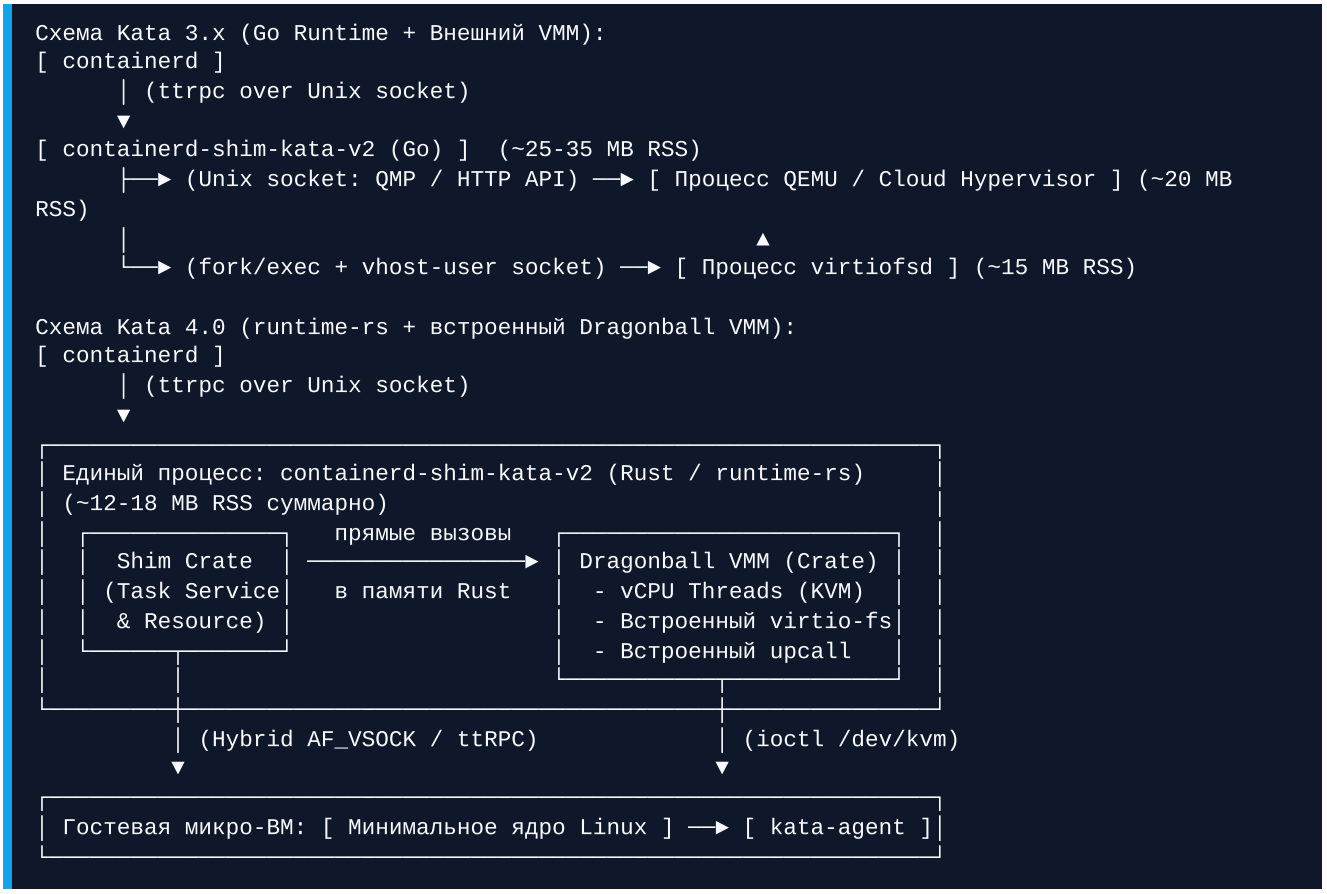
Однако в ветке 2.x и 3.x оставалась архитектурная проблема: `containerd-shim-kata-v2` был написан на языке Go (со сборщиком мусора и тяжелым рантаймом), а для запуска самой виртуальной машины он делал системные вызовы `fork()` и `exec()`, порождая два внешних процесса:

1. Внешний гипервизор (QEMU, Cloud Hypervisor или Firecracker), с которым шим общался через QMP-сокеты или REST API по Unix domain socket.
2. Внешний демон файловой системы `virtiofsd`, который общался с гипервизором по протоколу `vhost-user` через отдельный сокет.

2.3. Четвертое поколение (Kata 4.0, 2025–2026): Концепция **`Rust-first`** и единое адресное пространство

В 2025–2026 годах с выходом ветки **Kata Containers 4.0** завершился многолетний переход проекта на полностью переписанный на языке Rust рантайм — **`runtime-rs`** (исполняемый файл `containerd-shim-kata-v2`, собираемый из директории `src/runtime-rs`).

Сравнение процессной модели на хосте (Kata 3.x vs Kata 4.0 с Dragonball)



Почему переход на `runtime-rs` и встроенный гипервизор **Dragonball** стал революцией:

1. **Отсутствие `fork/exec` и сериализации команд управления:** поскольку гипервизор Dragonball скомпилирован как библиотека (Rust crate) прямо внутрь бинарного файла шима, для старта виртуальной машины больше не нужно порождать дочерний процесс ОС, парсить аргументы командной строки на 50 флагов и ждать инициализации сокетов управления. Шим вызывает методы структуры `DragonballHypervisor` напрямую в общей памяти.
2. **Встроенный сервер `virtio-fs`:** демон файловой системы больше не запускается отдельным процессом `virtiofsd`. Потоки обработки очередей `virtio-fs` работают внутри самого процесса `runtime-rs`, избавляя систему от накладных расходов протокола `vhost-user` и межпроцессной синхронизации.
3. **Предсказуемое потребление памяти без GC:** отказ от среды выполнения Go с ее сборщиком мусора (Garbage Collector) на стороне хоста и гостя (агент внутри ВМ также давно переписан на Rust) позволил снизить базовый оверхед памяти до ~12–15 МБ на под. При плотности в 200 подов на узел это экономит **от 6 до 9 гигабайт оперативной памяти** на каждом сервере.

Глава 3. Глубокое погружение в кодовую базу `runtime-rs` и гипервизор Dragonball

В репозитории `kata-containers/kata-containers` современный стек располагается в трех ключевых директориях: `src/runtime-rs` (хостовый рантайм), `src/dragonball` (встроенный VMM) и `src/agent` (гостевой агент). Разберем архитектуру каждого модуля.

3.1. Модульная структура `src/runtime-rs`

Кодовая база `runtime-rs` спроектирована на основе асинхронного рантайма **Tokio** и разделена на строго изолированные крейты (crates):

- **1. `crates/shim` и `crates/service`:** Точка входа бинарного файла. Реализует интерфейс `containerd::shim::asynchronous::Shim`. Когда `containerd` получает команду от `kubelet` (через CRI) или от демона Docker на создание контейнера, он запускает бинарник `containerd-shim-kata-v2` и

2. `crates/runtimes` (Менеджер жизненного цикла песочницы)

Содержит реализацию `VirtContainer` (виртуализированного контейнера). Здесь происходит критически важная логика разделения сущностей OCI:

- Когда Kubernetes создает Pod, сначала приходит запрос на запуск так называемого `pause`-контейнера (в аннотациях OCI передается `io.kubernetes.cri.container-type = "sandbox"`). `runtime-rs` распознает это событие, инициализирует объект `Sandbox`, запускает микро-ВМ и поднимает в ней `kata-agent`.
- Когда следом приходят запросы на запуск рабочих контейнеров приложения (`io.kubernetes.cri.container-type = "container"` с указанием `sandbox-id`), `runtime-rs` **не создает новую виртуальную машину**. Вместо этого он подключается к уже работающей микро-ВМ данного пода, монтирует корневую файловую систему нового контейнера внутрь существующей ВМ и командует `kata-agent` создать внутри гостевого ядра новые `namespaces` и `cgroups` для этого контейнера.

3. `crates/resource` (Управление ресурсами хоста и гостя)

Отвечает за подготовку и горячее подключение (hotplug) всех ресурсов до и во время работы ВМ:

- **`network`:** сканирует сетевое пространство имен (`netns`), созданное CNI-плагином (Calico, Cilium, Flannel) или Docker-бриджем. Внутри этого `netns` находится виртуальная пара `veth`. Поскольку гипервизор не может напрямую пробросить `veth`-интерфейс внутрь ВМ как PCI-устройство без бриджа, модуль `resource::network` создает в том же сетевом пространстве имен устройство **`tap`** (или **`macvtap`**) и связывает его с `veth` с помощью фильтра ядра **`tc`** **(Traffic Control redirect / mirred ingress/egress)** или eBPF-программы. Это позволяет пересылать Ethernet-кадры напрямую из `veth` в файловый дескриптор `/dev/net/tun`, который отдается гипервизору как бэкенд для `virtio-net`.
- **`rootfs` и `volume`:** анализирует тип хранилища контейнера. Если используется обычный `overlayfs` на хосте, путь пробрасывается через общий каталог `virtio-fs`. Если же контейнер использует блочное хранилище (например, `devicemapper`, Ceph RBD или локальный LVM-том), модуль `rootfs` пропускает `virtio-fs` и подключает блочное устройство напрямую в ВМ через драйвер **`virtio-blk`** или **`virtio-scsi`**, обеспечивая скорость работы с диском, идентичную "голому железу" (bare-metal).
- **`cgroups`:** создает на хосте контрольную группу для всего процесса микро-ВМ, чтобы гарантировать, что даже в случае компрометации гипервизора потребление CPU и памяти на хосте не превысит лимиты пода (`limits.cpu` / `limits.memory`).

4. `crates/hypervisor` (Единый трейт `Hypervisor`)

Абстрагирует взаимодействие с различными мониторами виртуальных машин (VMM) через асинхронный типаж (trait) Rust:

```
#[async_trait]
pub trait Hypervisor: Send + Sync {
    async fn prepare_vm(&self, id: &str, netns: Option<String>, annotations:
&HashMap<String, String>) -> Result<()>;
    async fn start_vm(&self, timeout: i32) -> Result<()>;
    async fn stop_vm(&self) -> Result<()>;
    async fn pause_vm(&self) -> Result<()>;
    async fn resume_vm(&self) -> Result<()>;
    async fn add_device(&self, device: DeviceType) -> Result<DeviceType>;
    async fn remove_device(&self, device: DeviceType) -> Result<()>;
    async fn update_device(&self, device: DeviceType) -> Result<()>;
    async fn get_agent_socket(&self) -> Result<String>;
    async fn disconnect(&self);
    async fn get_thread_ids(&self) -> Result<VcpuThreadIds>;
    async fn get_vmm_master_tid(&self) -> Result<u32>;
    async fn get_ns_path(&self) -> Result<String>;
    async fn check(&self) -> Result<()>;
    async fn get_jailer_root(&self) -> Result<String>;
    async fn save_state(&self) -> Result<HypervisorState>;
    async fn capabilities(&self) -> Result<Capabilities>;
```

3.2. Внутреннее устройство встроенного гипервизора Dragonball (`src/dragonball`)

Dragonball — это специализированный легковесный монитор виртуальных машин (MicroVM Monitor), изначально разработанный в Alibaba Cloud / Ant Group на базе компонентов открытого проекта **rust-vmm** (того же фундамента, на котором построены AWS Firecracker и Cloud Hypervisor) и переданный в кодовую базу Kata Containers.

В отличие от универсального QEMU (который насчитывает более 2 миллионов строк кода на C и эмулирует сотни устаревших устройств от дисководов гибких дисков до звуковых карт SoundBlaster 16 и контроллеров шины ISA), Dragonball написан на безопасном Rust и содержит минимально необходимый объем кода исключительно для запуска современных облачных ядер Linux по протоколу прямой загрузки (**Direct Kernel Boot** без BIOS/UEFI, либо через минимальный `rust-hypervisor-firmware`).

Ключевые архитектурные фичи Dragonball:

1. **Механизм Direct Kernel Upcall (Прямой вызов ядра без шины PCI ACPI):** В классических гипервизорах (QEMU, Cloud Hypervisor) для того, чтобы на горячую подключить в работающую виртуальную машину новое ядро процессора (vCPU Hotplug), планку памяти (Memory Hotplug) или сетевую карту, гипервизор должен эмулировать контроллер **ACPI (Advanced Configuration and Power Interface)**, генерировать аппаратное прерывание SCI, а гостевое ядро должно выполнить байт-код таблиц AML (ACPI Machine Language). Это медленно, требует наличия сложного кода эмуляции ACPI в VMM и увеличивает поверхность атаки. В Dragonball реализован механизм **`upcall`** — прямой канал связи между VMM и драйвером в гостевом ядре поверх очередей `virtio` / `vsock` . Когда хосту нужно добавить vCPU или устройство, Dragonball отправляет компактную структуру данных напрямую в гостевое ядро через `upcall` -клиент, и ядро мгновенно инициализирует ресурс за единицы миллисекунд без участия подсистемы ACPI.
2. **Управление виртуальной памятью (`vm-memory` и `virtio-mem`):** Когда вы задаете контейнеру лимит памяти в 8 ГБ (`docker run -m 8g`), гипервизор не должен сразу резервировать 8 ГБ физической оперативной памяти на хосте. Dragonball выделяет область виртуального адресного пространства через `mmap(..., MAP_ANONYMOUS | MAP_NORESERVE)` и регистрирует ее в KVM через системный вызов `ioctl(KVM_SET_USER_MEMORY_REGION)` . Физические страницы на хосте выделяются динамически (on-demand paging) по мере того, как процесс внутри контейнера реально обращается к ним. При освобождении памяти внутри контейнера с помощью **`virtio-balloon`** или **`virtio-mem`** страницы возвращаются обратно хостовой ОС через вызов `madvise(..., MADV_DONTNEED)` .
3. **Встроенный многопоточный движок `dbs-virtio-devices` (`virtio-fs` с поддержкой DAX):** Внутри крейта Dragonball интегрирован собственный сервер `virtio-fs` . Он отображает корневую файловую систему контейнера с хоста напрямую в окно физического адресного пространства гостя (PCI Base Address Register — BAR). Когда приложение внутри контейнера читает файл (например, загружает разделяемую библиотеку `.so` или веса нейросети), гостевое ядро не копирует данные в собственную оперативную память ВМ. Благодаря технологии **DAX (Direct Access)** таблицы страниц гостевого процессора (EPT — Extended Page Tables на Intel или NPT на AMD) настраиваются так, что виртуальный адрес внутри контейнера напрямую указывает на страницу в Page Cache ядра хоста! **Эффект:** если на одном сервере запущено 100 изолированных микро-ВМ Kata с одинаковым базовым образом (например, `python:3.12` или `ubuntu:24.04`), все 100 виртуальных машин читают одни и те же физические страницы памяти с разделяемыми библиотеками из кэша хоста, не дублируя их в RAM каждой ВМ.

3.3. Анатомия `kata-agent` (`src/agent`) внутри гостевой микро-ВМ

Что происходит внутри самой виртуальной машины в момент старта?

1. Гипервизор загружает в память образ сжатого или несжатого ядра Linux (`vmlinuz` / `vmlinux`) и минимальный образ корневой файловой системы: либо `initrd` (Initial RAM Disk, распаковываемый в память), либо `rootfs.img` (образ файловой системы только для чтения, подключаемый через `virtio-blk` или `pmem` с флагом DAX, чтобы не тратить оперативную память на хранение файлов самой гостевой ОС).
2. Ядро Linux инициализирует базовые подсистемы за 30–50 мс и запускает в качестве процесса **`PID 1`** (вместо тяжелого `systemd`) специализированный бинарный файл, написанный на Rust — **`kata-agent`** .

Глава 4. Практическое руководство: Установка, конфигурация и отладка

Перейдем к практической части: как развернуть современный стек Kata Containers на Linux-сервере, настроить его для работы с `containerd` и Docker, а также тонко сконфигурировать параметры гипервизора в файле `configuration.toml`.

4.1. Проверка аппаратных требований хоста

Перед установкой необходимо убедиться, что процессор поддерживает аппаратную виртуализацию (флаги `vmx` для Intel или `svm` для AMD), а модуль ядра `kvm` и устройство `vhost-vsock` активны. На облачных VDS/VPS (например, в Selectel или immers.cloud) для этого должна быть включена поддержка **Nested Virtualization** (вложенной виртуализации).

```
# 1. Проверка наличия флагов виртуализации процессора
grep -E --color 'vmx|svm' /proc/cpuinfo | head -n 5

# 2. Проверка загруженных модулей KVM и VSOCK
lsmod | grep -E 'kvm|vhost'

# Если модули vhost_vsock и vhost_net не загружены, активируем их:
sudo modprobe vhost_vsock
sudo modprobe vhost_net

# Записываем в автозагрузку при старте системы:
cat <<EOF | sudo tee /etc/modules-load.d/kata-modules.conf
vhost_vsock
vhost_net
EOF
```

4.2. Установка и структура артефактов Kata Containers (ветка 3.x / 4.x)

Поскольку пакеты в стандартных репозиториях Ubuntu/Debian (`apt install kata-containers`) сильно отстают от апстрима, в продакшене и лабораторных средах используется официальный пакетный релиз из GitHub (устанавливается в изолированный префикс `/opt/kata`) либо автоматизированный скрипт `kata-manager.sh`.

```
# Установка последней стабильной версии через официальный скрипт kata-manager:
bash -c "$(curl -fsSL https://raw.githubusercontent.com/kata-containers/kata-
containers/main/utils/kata-manager.sh)"

# Либо ручная распаковка статического тарбола релиза в /opt/kata:
KATA_VERSION="3.12.0" # или 4.0.0 для ветки с runtime-rs по умолчанию
curl -LO "https://github.com/kata-containers/kata-
containers/releases/download/${KATA_VERSION}/kata-static-${KATA_VERSION}-amd64.tar.xz"
sudo tar -xvf "kata-static-${KATA_VERSION}-amd64.tar.xz" -C /

# Проверка готовности хоста встроенной диагностической утилитой:
sudo /opt/kata/bin/kata-runtime check
# Вывод покажет: System is capable of running Kata Containers
```

После распаковки в директории `/opt/kata/` появляется полная автономная среда со своими скомпилированными гипервизорами, ядрами и конфигурациями:

- `/opt/kata/bin/containerd-shim-kata-v2` — классический Go-шим.
- `/opt/kata/runtime-rs/bin/containerd-shim-kata-v2` — новый Rust-шим (`runtime-rs` со встроенным Dragonball).
- `/opt/kata/bin/qemu-system-x86_64` и `/opt/kata/bin/cloud-hypervisor` — скомпилированные с нужными флагами безопасности бинарники гипервизоров.
- `/opt/kata/libexec/virtiofsd` — демон файловой системы (для внешних гипервизоров).
- `/opt/kata/share/kata-containers/vmlinux.container` — оптимизированное гостевое ядро Linux.

4.3. Интеграция с `containerd` и Docker Engine

Создадим символические ссылки в `/usr/local/bin`, чтобы `containerd` и Docker могли вызывать разные профили Kata (например, профиль на базе Cloud Hypervisor и профиль на базе нового Dragonball), а также настроим `/etc/containerd/config.toml` (Версия схемы 2) и `/etc/docker/daemon.json`:

```
# 1. Создание символических ссылок для стандартного Go-шима и нового Rust-шима (kata-rs):
sudo ln -sf /opt/kata/bin/containerd-shim-kata-v2 /usr/local/bin/containerd-shim-kata-v2
sudo ln -sf /opt/kata/runtime-rs/bin/containerd-shim-kata-v2 /usr/local/bin/containerd-shim-
kata-rs-v2

# 2. Фрагмент /etc/containerd/config.toml (регистрация нескольких обработчиков рантаймов):
version = 2
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes]
  # Стандартный рантайм по умолчанию (runc)
  [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc]
    runtime_type = "io.containerd.runc.v2"

  # Kata с внешним гипервизором Cloud Hypervisor
  [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.kata-clh]
    runtime_type = "io.containerd.kata.v2"
    privileged_without_host_devices = true
    pod_annotations = ["io.katacontainers.*"]
    [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.kata-clh.options]
      ConfigPath = "/opt/kata/share/defaults/kata-containers/configuration-clh.toml"

  # Kata 4.0 на базе Rust-рантайма (runtime-rs) и встроенного Dragonball VMM
  [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.kata-dragonball]
    runtime_type = "io.containerd.kata-rs.v2"
    privileged_without_host_devices = true
    pod_annotations = ["io.katacontainers.*"]
    [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.kata-dragonball.options]
      ConfigPath = "/opt/kata/share/defaults/kata-containers/configuration-dragonball.toml"

# 3. Конфигурация /etc/docker/daemon.json для прямого использования флага --runtime в
Docker:
{
  "runtimes": {
    "kata": { "runtimeType": "io.containerd.kata.v2" },
    "kata-rs": { "runtimeType": "io.containerd.kata-rs.v2" }
  }
}

# 4. Перезапуск служб и проверка изоляции ядра:
sudo systemctl restart containerd docker
uname -r                                           # На хосте: 6.8.0-45-generic
docker run --rm --runtime=kata-rs ubuntu:24.04 uname -r # Внутри Kata: 6.6.52-kata-custom
```

ПРИМЕЧАНИЕ К ПАРАМЕТРУ PRIVILEGED_WITHOUT_HOST_DEVICES = TRUE

По умолчанию, если запустить контейнер с флагом `--privileged`, `containerd` попытается пробросить внутрь контейнера **все** устройства хоста из `/dev`. Для виртуальной машины это недопустимо и нарушает безопасность. Флаг `privileged_without_host_devices = true` говорит `containerd`: выдай процессу внутри микро-ВМ все права root (`CAP_SYS_ADMIN` и др. относительно **гостевого** ядра), но не пробрасывай физические блочные устройства хоста. Это позволяет безопасно запускать **Docker-in-Docker (DinD)** или загружать модули ядра внутри контейнера без риска для хоста!

4.4. Анатомия конфигурационного файла `configuration.toml` (Тонкий тюнинг) и аннотации

Конфигурационный файл Kata Containers разделен на несколько секций: `[hypervisor.<name>]`, `[agent.kata]`, `[runtime]`. В реальной эксплуатации разным контейнерам нужны разные параметры ВМ: вместо создания десятка разных файлов `configuration.toml` Kata позволяет передавать параметры прямо в манифесте пода Kubernetes через префикс `io.katacontainers.config.hypervisor.*`.

```
# ============================================================================
# СЕКЦИЯ ГИПЕРВИЗОРА (configuration.toml на примере dragonball / clh / qemu)
# ============================================================================
[hypervisor.dragonball]
kernel = "/opt/kata/share/kata-containers/vmlinux.container"
image = "/opt/kata/share/kata-containers/kata-containers.img"
kernel_params = "systemd.unit=kata-containers.target systemd.mask=systemd-networkd.service
systemd.mask=systemd-networkd.socket scsi_mod.scan=none"

# Базовое число vCPU при старте микро-ВМ (при --cpus=4 Kata сделает hotplug 3 vCPU на лету)
default_vcpus = 1
default_maxvcpus = 16

# Базовый объем RAM микро-ВМ (в МБ), если контейнеру не задан лимит памяти (-m)
default_memory = 2048

# Механизм проброса ФС: "virtio-fs", "virtio-fs-nydus", "inline-virtio-fs" (для Dragonball)
shared_fs = "inline-virtio-fs"
virtio_fs_cache_size = 1024   # Размер окна памяти DAX (Direct Access) в МБ
virtio_fs_cache = "auto"
hotplug_vfio_on_root_bus = true

[agent.kata]
debug_console_enabled = true  # Позволяет зайти внутрь самой ВМ с хоста
dial_timeout = 10

[runtime]
enable_debug = false
internetworking_model = "tcfilter" # Перехват netns с помощью tc redirect
enable_annotations = ["default_memory", "default_vcpus", "kernel", "kernel_params"]

# ============================================================================
# ПРИМЕР МАНИФЕСТА KUBERNETES С ПЕРЕОПРЕДЕЛЕНИЕМ РЕСУРСОВ ЧЕРЕЗ АННОТАЦИИ
# ============================================================================
apiVersion: v1
kind: Pod
metadata:
  name: custom-vm-pod
  annotations:
    io.katacontainers.config.hypervisor.default_memory: "8192"
    io.katacontainers.config.hypervisor.default_vcpus: "4"
    io.katacontainers.config.hypervisor.kernel: "/opt/kata/share/kata-containers/vmlinux-
gpu-6.8"
spec:
  runtimeClassName: kata-dragonball
  containers:
    - name: worker
      image: ubuntu:24.04
      command: ["sleep", "infinity"]
```

Глава 5. Изоляция AI-агентов, проброс GPU и Confidential Computing (CoCo)

В 2025–2026 годах главный вектор развития Kata Containers сместился в сторону искусственного интеллекта: 1) Безопасное выполнение кода автономными AI-агентами (AI Sandboxing); 2) Аппаратный проброс GPU внутрь защищенных анклавов (Confidential AI / Trusted Execution Environments).

5.1. Почему автономным AI-агентам необходимы микро-ВМ Kata

Современные LLM-системы (Code Interpreters, автономные SWE-агенты для написания и отладки кода, агенты анализа данных) в автоматическом режиме скачивают произвольные пакеты из PyPI, npm или Cargo, компилируют C/C++/Rust расширения, запускают сгенерированные Python/Bash-скрипты и делают сетевые запросы. С точки зрения ИБ, **любой код, сгенерированный большой языковой моделью (особенно при обработке внешних данных, подверженных атакам Prompt Injection), является на 100% враждебным и недоверенным**. В обычных контейнерах (`runc`) Indirect Prompt Injection может заставить AI-агента выполнить эксплойт ядра Linux, а `gVisor` часто ломает работу научных Python-библиотек (PyTorch, NumPy, Triton/Numba).

1. На каждый сеанс работы агента запускается микро-ВМ под управлением `runtime-rs` и Dragonball.
2. Для устранения задержек холодного старта применяется **VM Templating / Snapshot Restore** (на базе Cloud Hypervisor или QEMU): базовая микро-ВМ с уже загруженным в память Python и тяжелыми библиотеками (`torch`, `pandas`) ставится на паузу, ее состояние памяти сохраняется в файл с отображением `MAP_PRIVATE` (Copy-on-Write). Новые песочницы клонируются из снимка за **5–15 миллисекунд**.
3. Сетевой доступ микро-ВМ жестко режется на уровне хостового `tap`-интерфейса через eBPF (разрешается доступ только к внутреннему прокси-серверу репозиториев).

5.2. Проброс физических GPU и MIG-инстансов (VFIO Passthrough)

Внутри контейнера Kata работает **отдельное ядро Linux**, в котором нет доступа к `/dev/nvidia*` хоста. Поэтому используется аппаратный проброс шины PCIe через подсистему ядра **VFIO (Virtual Function I/O)** с использованием аппаратного блока **IOMMU** (`Intel VT-d` или `AMD-Vi`), гарантирующего защиту DMA-обращений памяти.

- **1. Полный проброс физической карты (Full GPU Passthrough):** на хосте видеокарта отвязывается от драйвера `nvidia` и привязывается к `vfio-pci`. При старте пода Kata подключает PCI-устройство напрямую в виртуальную шину PCIe микро-ВМ, где работает гостевой драйвер NVIDIA.
- **2. Виртуализация и нарезка GPU (SR-IOV + NVIDIA MIG):** на ускорителях Ampere (A100), Hopper (H100/H200) и Blackwell (B200) карту можно аппаратно разделить до **7 независимых инстансов MIG**. В сочетании с режимом SR-IOV каждый MIG-инстанс представляется на шине PCIe хоста как отдельная виртуальная функция (**VF**), пробрасываемая в отдельную микро-ВМ через `vfio-pci`.

```
# 1. В параметрах загрузки ядра (/etc/default/grub) включаем IOMMU (intel_iommu=on iommu=pt)
# 2. Узнаем PCI ID и Vendor:Device ID видеокарты и привязываем к драйверу vfio-pci:
lspci -nn | grep -i nvidia
# Вывод: 01:00.0 3D controller [0302]: NVIDIA Corporation GA100 [A100 PCIe 80GB] [10de:20b5]
sudo modprobe vfio-pci
echo "10de 20b5" | sudo tee /sys/bus/pci/drivers/vfio-pci/new_id

# 3. В файле configuration-clh.toml активируется поддержка горячего подключения VFIO:
[hypervisor.clh]
kernel = "/opt/kata/share/kata-containers/vmlinux-nvidia-gpu.container"
image = "/opt/kata/share/kata-containers/kata-containers-nvidia-gpu.img"
hotplug_vfio_on_root_bus = true
pcie_root_port = 4
```

5.3. Confidential Containers (CoCo) и защита весов AI-моделей (Trusted AI)

Для защиты проприетарных LLM-моделей и данных от администратора публичного облака (имеющего права `root` на хосте) Kata Containers объединился с инициативой CNCF **Confidential Containers (CoCo)**:

- **На уровне CPU (AMD SEV-SNP / Intel TDX):** аппаратный контроллер памяти процессора на лету шифрует каждую страницу RAM микро-ВМ уникальным эфемерным ключом AES-128/256, недоступным ни ядру хоста, ни гипервизору.
- **На уровне GPU (NVIDIA Confidential Computing на H100/H200/Blackwell):** между зашифрованной памятью микро-ВМ и видеокартой устанавливается защищенный аппаратный канал по шине PCIe (протокол **SPDM**). Драйвер внутри ВМ шифрует тензоры и веса модели перед отправкой по PCIe, а расшифровка происходит внутри защищенного периметра GPU.
- **Удаленная аттестация (Remote Attestation) и загрузка образов внутри гостя:** хост запускает пустую зашифрованную микро-ВМ. Внутри нее рядом с `kata-agent` работают Rust-компоненты **`image-rs`** и **`attestation-agent`**. Агент формирует криптографический отчет оборудования (Hardware Quote) и отправляет его на доверенный сервер ключей (**KBS — Key Broker Service**). После сверки хэшей регистров `RTMR` KBS передает ключ напрямую внутрь микро-ВМ, и `image-rs` скачивает и расшифровывает слои контейнера и веса модели исключительно внутри зашифрованной памяти ВМ!

Глава 6. Сетевая подсистема и подсистема хранения данных: устройство изнутри

6.1. Сетевой путь пакета (Network Data Path) и три модели моста (**`internetworking_model`**)

1. **`tcfilter`** **(Стандарт для продакшена):** рантайм Kata заходит в созданный CNI `netns`, создает виртуальный интерфейс `tap0` и устанавливает в подсистему `tc` (Traffic Control) ядра Linux два фильтра перенаправления (`act_mirred`) между `eth0` и `tap0` на уровне L2. Модуль ядра **`vhost-net`** выносит обработку колец `virtqueues` в ядерный поток `vhost-<pid>` без переключений контекста User-space <-> Kernel-space.
2. **`macvtap`:** создает устройство `macvtap` напрямую поверх основного интерфейса контейнера (чуть меньшая задержка, но есть ограничения с некоторыми CNI).
3. **SR-IOV VFIO Network Passthrough:** на картах Intel XXV710/E810 или NVIDIA Mellanox ConnectX-6/7 виртуальная функция (VF) пробрасывается напрямую в микро-ВМ через `vfio-pci`, полностью минуя ядро хоста, `veth`, `tap` и `iptables`.

6.2. Подсистема хранения: **`virtio-fs`** vs **`Devicemapper / Block`** vs **`Nydus`**

- **Паттерн 1 (Классический `virtio-fs` поверх `overlayfs` хоста):** универсальный вариант через протокол FUSE поверх `virtio`, но при активном `chown` или создании сотен тысяч мелких файлов (`npm install`) дает оверхед на метаданных.
- **Паттерн 2 (Блочный снапшоттер `devmapper` / `blockfile`):** каждый контейнер получает блочное устройство `/dev/mapper/container-xyz` (LVM thin-pool), подключаемое в ВМ напрямую через драйвер **`virtio-blk`** в обход FUSE на скорости **98–99% от bare-metal**.
- **Паттерн 3 (Ленивая загрузка образов через `Nydus RAFS v6` + `EROFS`):** для старта контейнера реально читается лишь 6–8% данных образа. Контейнер стартует мгновенно с крошечным индексом метаданных, а блоки данных (chunks) подкачиваются по сети на лету при обращении к страницам файла через `EROFS` и `fscache`.

Глава 7. Структура следующих частей руководства (Части 2 и 3)

- **Часть 2 (Главы 7–9 — Kubernetes и продакшен):** оператор `kata-deploy`, матрица `RuntimeClass`, `PodOverhead`; архитектура Microsoft AKS Pod Sandboxing, IBM Cloud и Ant Group (мониторинг через eBPF и cAdvisor); изолированные SaaS CI/CD-раннеры без `privileged` режима.
- **Часть 3 (Главы 10–12 — Кастомная сборка ядра и отладка):** сборка ядра (`build-kernel.sh`) с eBPF/WireGuard/ZFS и `rootfs` (`osbuilder`); отладка через `kata-runtime exec`, трассировка `ttRPC`, профилирование `perf kvm` и сводные бенчмарки (CPU, RAM, FIO, iperf3).