ТЕХНИЧЕСКОЕ РУКОВОДСТВО • ЧАСТЬ 2 (ПРОМЫШЛЕННАЯ ЭКСПЛУАТАЦИЯ)

KATA CONTAINERS В KUBERNETES И PRODUCTION (2025–2026)

Масштабирование изолированных песочниц: развертывание через `kata-deploy`, настройка `PodOverhead`, архитектура AKS / IBM Cloud и безопасные CI/CD ранеры

Глава 7. Интеграция Kata Containers с Kubernetes (Production Guide)

7.1. Установка через оператор `kata-deploy` и настройка `RuntimeClass`

В продакшн-кластерах Kubernetes ручная установка бинарников через скрипты не применяется. Стандартом де-факто является использование официального DaemonSet-оператора **`kata-deploy`**, который автоматически распространяет бинарники рантайма, ядра, образы гостевой ФС и конфигурации на все узлы кластера (или на узлы с определенными лейблами).

Для развертывания применяется манифест оператора:

```
# Применение официального манифеста ката-оператора
kubectl apply -f https://raw.githubusercontent.com/kata-containers/kata-
containers/main/tools/packaging/kata-deploy/k8s/kata-deploy.yaml

# Проверка статуса развертывания демона на узлах:
kubectl get pods -n kube-system -l name=kata-deploy
```

После успешной установки оператор создает в кластере объекты типа **`RuntimeClass`**. Они позволяют разработчикам указывать в манифесте подов, какой именно рантайм использовать (`runc` для обычных приложений или `kata-dragonball` / `kata-clh` для изолированных нагрузок).

```
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: kata-dragonball
handler: kata-dragonball
scheduling:
  nodeSelector:
    matchLabels:
      node.kubernetes.io/instance-type: "compute-optimized"
  tolerations:
    - key: "kata.katacontainers.io/ops"
      operator: "Exists"
      effect: "NoSchedule"
```

7.2. Учет накладных расходов памяти через `PodOverhead`

Важнейшая проблема при планировании ресурсов в Kubernetes с Kata Containers — **оверхед памяти и CPU на обслуживание самой микро-ВМ**. Если вы запускаете в поде контейнер с лимитом `memory: 512Mi`, а планировщик Kube-scheduler учтет только эти 512 МБ, узел быстро упадет по OOM, так как гостевое ядро Linux, процесс шима `runtime-rs` и кэш `virtio-fs` требуют еще около **35–60 МБ** (для Go-версий) или **15–20 МБ** (для Kata 4.0 с Dragonball) дополнительной RAM.

Чтобы планировщик корректно резервировал ресурсы, в объект `RuntimeClass` или непосредственно в манифест пода добавляется секция **`PodOverhead`** (в современных версиях Kata оператор настраивает это автоматически):

```
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
```

Глава 8. Архитектура Kata Containers в публичных облаках и корпоративных платформах

8.1. Microsoft Azure Kubernetes Service (AKS Pod Sandboxing)

В Microsoft Azure функция **AKS Pod Sandboxing** построена на базе Kata Containers и гипервизора **Cloud Hypervisor**. Архитектурное решение позволяет безопасно запускать код сторонних клиентов (мультитенантность) на одном общем кластере Kubernetes без риска перехвата данных:

- **Изоляция на уровне хоста Hyper-V / Azure Linux:** Узлы AKS работают на базе оптимизированного дистрибутива Azure Linux. При запросе пода с `runtimeClassName: azure-kata` узел поднимает микро-ВМ через Cloud Hypervisor с аппаратной изоляцией.
- **Оптимизация сетевых задержек:** Интеграция с Azure CNI позволяет напрямую привязывать сетевые интерфейсы через ускоренную сетевую подсистему Azure Accelerated Networking к гостевым микро-ВМ без деградации пропускной способности.

8.2. Опыт IBM Cloud и Ant Group (Масштабирование и Observability)

Крупнейшие контрибьюторы Kata Containers (Ant Group и IBM Cloud) эксплуатируют сотни тысяч микро-ВМ в режиме 24/7. Главные уроки продакшена:

- **Проблема Observability (Мониторинг внутри ВМ):** Стандартный kubelet собирает метрики cAdvisor с хоста через `cgroups`. В Kata Containers процессы внутри контейнера скрыты внутри гостевой ВМ. Для решения этой проблемы `kata-agent` транслирует гостевые метрики памяти и CPU наружу через протокол `ttRPC`, позволяя Prometheus и Datadog собирать точную телеметрию на уровне приложений.
- **Сбор логов (`stdout` / `stderr`):** Потоки вывода контейнера перехватываются гостевым агентом и через сокеты `vsock` передаются хостовому шиму, откуда их стандартным образом забирает Fluentbit или Vector.

Глава 9. Построение полностью безопасных SaaS CI/CD-ранеров на базе Kata Containers

9.1. Проблема безопасности традиционных CI/CD ранеров (`DinD`)

Классические CI/CD системы (GitLab CI, GitHub Actions Runners, Woodpecker CI) часто требуют запуска сборочных контейнеров в режиме `privileged` или проброса сокета хостового демона Docker (`/var/run/docker.sock`) внутрь ранера для сборки докер-образов через механизм **Docker-in-Docker (DinD)**.

Вектор атаки: Сборка недоверенного кода из публичного PR в GitHub, содержащего вредоносный скрипт в `Dockerfile` или `Makefile`, позволяет злоумышленнику мгновенно получить полный доступ к хосту ранера и украсть секреты кластера (SSH-ключи, токены AWS/K8s) через `docker.sock`.

9.2. Решение на базе Kata Containers и беспривилегированных сборок

Использование Kata Containers (профиль `kata-dragonball` или `kata-clh`) с флагом `privileged_without_host_devices = true` в `containerd` полностью закрывает этот вектор:

- Ранер исполняется внутри изолированной микро-ВМ со своим собственным ядром Linux.
- Сборка образов через Buildah, Kaniko или Docker-in-Docker внутри такой микро-ВМ выполняется абсолютно безопасно: даже при успешном взломе ядра гостевой ВМ эксплойт не затронет физический узел и соседние пайплайны других команд.

ТЕХНИЧЕСКОЕ РУКОВОДСТВО • ЧАСТЬ 3 (СБОРКА И ОТЛАДКА)

КАСТОМНАЯ СБОРКА ЯДРА, ОБРАЗОВ И НИЗКОУРОВНЕВАЯ ОТЛАДКА KATA

Создание собственного гостевого ядра с модулями eBPF/WireGuard, сборка `rootfs` через osbuilder и диагностика через `kata-runtime exec`

Глава 10. Кастомная сборка гостевого ядра Linux и образов rootfs

10.1. Зачем собирать кастомное гостевое ядро?

Стандартное ядро Kata (`vmlinux.container`) оптимизировано под минимальный размер и швидкий старт, поэтому в нем отключены многие редко используемые модули. Если вашему приложению внутри контейнера требуются специфические подсистемы (например, современный **eBPF** с BTF, модуль VPN **WireGuard**, файловая система **ZFS** или специфические сетевые драйверы), стандартное ядро вернет ошибку.

10.2. Процесс сборки через скрипты `osbuilder` и репозиторий ядра

Исходный код конфигураций гостевых ядер хранится в репозитории Kata. Для сборки кастомного ядра используется официальная утилита `kata-kernel-builder` :

```
# Клонирование репозитория сборщика ядер Kata
git clone https://github.com/kata-containers/kata-containers.git
cd kata-containers/tools/packaging/kernel-builder

# Генерация конфига и включение нужных опций (например, CONFIG_BING=y, CONFIG_WIREGUARD=m)
make kernel-config KERNEL_VERSION=6.8.x

# Запуск сборки оптимизированного гостевого ядра:
make build-kernel KERNEL_VERSION=6.8.x
# На выходе получаем готовый файл vmlinux-custom, который прописываем в configuration.toml
```

10.3. Сборка минимального корневого образа (`osbuilder`)

Для создания кастомного образа гостевой файловой системы (`kata-containers.img`) с предустановленными системными утилитами используется утилита `osbuilder` :

```
cd ../osbuilder
# Сборка образа для Ubuntu/Debian базового дистрибутива:
sudo ./scripts/build-rootfs.sh -d ubuntu
# Генерация финального .img диска для гипервизора:
sudo ./scripts/image-builder.sh -r rootfs-ubuntu.tar
```

Глава 11. Низкоуровневая отладка и бенчмарки производительности

11.1. Методика отладки: Вход в микро-ВМ и трассировка ttRPC

Если контейнер внутри Kata падает при старте или ведет себя некорректно, стандартный лог `kubectl logs` может не показать причину сбоя внутри гостевой ОС. Для прямой диагностики используется отладочная консоль:

```
# 1. Убедитесь, что в configuration.toml включен параметр: debug_console_enabled = true
# 2. Подключение к работающей микро-ВМ пода с хоста через отладочный сокет:
sudo /opt/kata/bin/kata-runtime exec <container-id> sh

# Проверка процессов внутри гостевой ВМ (мы увидим kata-agent как PID 1 и процессы
пользователя):
ps faux
```

11.2. Сводные бенчмарки производительности (Kata 4.0 vs Runc vs gVisor)

Тестирование производительности современных версий Kata Containers (4.0 с Dragonball) по сравнению с нативным `runc` и `gVisor` демонстрирует следующие показатели:

Тип нагрузки (Benchmark)	Стандартный `runc`	Google `gVisor` (`runsc`)	Kata 4.0 (Dragonball)
CPU Compute (Sysbench Prime)	100% (Эталон)	97.5%	**99.6%** (нативная скорость KVM)
Memory Bandwidth (Stream)	100% (Эталон)	94.0%	**98.9%** (прямой доступ к RAM)
Disk I/O (FIO Random Read/Write)	100% (Эталон)	55–70% (оверхед Sentry)	**97.5%** (через virtio-fs DAX / virtio-blk)
Network Throughput (iperf3)	100% (Эталон)	75–85%	**98.2%** (vhost-net + tcfilter)
Cold Start Time (Холодный старт)	~25 мс	~65 мс	**~75–95 мс** (Dragonball без fork/exec)

ЗАКЛЮЧЕНИЕ КНИГИ

Kata Containers 4.0 превратилась из тяжелого экспериментального решения в высокопроизводительный стандарт безопасной изоляции облачных нативных приложений, AI-агентов и конфиденциальных вычислений. Переход на Rust-архитектуру (`runtime-rs` + Dragonball) снял исторические проблемы с потреблением памяти и задержками старта, сделав микро-ВМ идеальным выбором для мультитенантных Kubernetes-кластеров.