Неделя 14. Лекция 7. Мониторинг: метрики, дашборды, алерты

Часть 2 из 2. Начало лекции — в материалах предыдущей недели.

11. SLI, SLO и бюджет ошибок

Чтобы спорить не о впечатлениях («что-то медленно»), договариваются о числах.

- **SLI** (индикатор) — измеряемая доля хороших событий: доля запросов быстрее 500 мс, доля ответов не 5xx.
- **SLO** (цель) — какой SLI считаем нормой за период: «99% запросов быстрее 500 мс и меньше 1% ответов 5xx за 30 дней».
- **Бюджет ошибок** — разрешённый остаток: 1%. На миллион запросов за месяц это 10 000 медленных или упавших. Пока бюджет есть, команда выкатывает новое; когда он съеден, приоритет — надёжность.

SLI для нашего сервиса можно посчитать из той же гистограммы. Границы 500 мс в корзинах нет, ближайшая — `le="0.512"`, её и берём:

```
sum(rate(http_request_duration_seconds_bucket{job="demo-api", le="0.512"}[5m]))
  / sum(rate(http_request_duration_seconds_count{job="demo-api"}[5m]))
```

Результат 0,993 означает, что 99,3% запросов уложились в 512 мс. Выбирая SLO, подгоняйте порог к границам корзин — иначе считать придётся интерполяцией.

SLO меняет и подход к алертам. Вместо «p95 больше секунды 30 секунд подряд» зрелые команды алертят на **скорость сжигания бюджета**: если в таком темпе месячный бюджет кончится за пару дней, это повод будить человека, а короткий всплеск, съевший долю процента, — нет. На стенде мы используем простые пороги, но смысл тот же: алерт привязан к тому, что обещано пользователю.

12. Дашборды

Grafana сама данных не хранит. У неё есть **источник данных** — у нас это Prometheus, заданный файлом `grafana/provisioning/datasources/prometheus.yml` с адресом `http://prometheus:9090` (имя сервиса в сети compose-проекта). **Панель** — один запрос PromQL и способ его показать: график, число, таблица. **Дашборд** — набор панелей с общим интервалом времени.

Наш дашборд `DemoApi` (`uid: demo-api`, окно 15 минут, обновление раз в 5 с):

Панель	Запрос	Вопрос
API up	`up{job="demo-api"}`	Жив ли сервис
RPS по кодам ответа	`sum by (code) (rate(http_requests_received_total{job="demo-api"}[1m]))`	Сколько запросов и с какими кодами (R и E)
Latency p95	`histogram_quantile(0.95, sum by (le) (rate(http_request_duration_seconds_bucket{job="demo-api"}[1m])))`	Как быстро (D)
Память контейнера / лимит	`container_memory_working_set_bytes{name="demo-api-1"}` и `container_spec_memory_limit_bytes{name="demo-api-1"}`	Далеко ли до OOM
CPU контейнеров (ядра)	`sum by (name) (rate(container_cpu_usage_seconds_total{name=~"demo-.*"}[1m]))`	Нагрузка на сервис и БД

Порядок панелей не случаен и повторяет ход мысли дежурного. Сверху «жив ли», ниже RED — то, что чувствует пользователь, внизу ресурсы, которые объясняют отклонения. У каждой панели должна быть единица измерения: запросы в секунду, секунды, байты. Без неё «0.25» на панели p95 можно прочитать и как 250 мс, и как 0,25 мс. Порог алерта полезно нарисовать на той же панели линией: тогда видно не только «растёт», но и «насколько близко к срабатыванию».

Переменные делают дашборд интерактивным: например, переменная `action` со значениями из `label_values(http_requests_received_total{job="demo-api"}, action)` и фильтр `action="$action"` в запросах. В нашем дашборде переменных нет — сервис один, — но на общих дашбордах по многим сервисам без них не обойтись.

Дашборд как код. JSON лежит в `monitoring/grafana/dashboards/`, файл `provisioning/dashboards/default.yml` велит Grafana загрузить всё из этой папки в папку *DevOps course*. Grafana периодически перечитывает файлы, поэтому изменение в git доезжает до неё без кликов. Provisioned-дашборд нельзя сохранить из UI: правка делается в UI, экспортируется в JSON и коммитится через PR. Так дашборд переживает пересоздание Grafana и проходит ревью, как код.

Что видно под нагрузкой. `scripts/load.ps1` в цикле создаёт задачу (`Tasks/Create`, 201), переводит её в `InProgress` (`Tasks/ChangeStatus`, 200) и запрашивает список (`Tasks/List`, 200), с паузой 300 мс. На панели RPS появляются ровные линии 200 и 201, p95 держится в десятках миллисекунд, память стоит на месте. Это «норма» — без неё отклонение не с чем сравнить.

Сценарий из лабы	Что меняется на дашборде
`/api/chaos/error` раз в секунду	На RPS появляется линия `500`; доля 5xx выходит за 5%
`/api/chaos/slow?ms=1500` подряд	Растёт p95, но только если медленных запросов больше 5% от всех
`POST /api/chaos/crash`	Провал на RPS, `API up` может мигнуть в 0, счётчики начинаются заново
`POST /api/chaos/leak`	Working set ступенями поднимается к линии лимита, после OOM падает

Строка про `slow` показательна: p95 реагирует на долю медленных запросов, а не на их факт. Одиночный медленный запрос среди сотни быстрых p95 не сдвинет — для этого нужен p99 или отдельный запрос по действию.

> **Демонстрация для преподавателя.** Запустите `load.ps1`, откройте дашборд и Prometheus *Query* рядом. Выполните `http_requests_received_total{action="Create"}` в табличном виде — число растёт. Затем `rate(...[1m])` — ровная линия. Затем вызовите `POST /api/chaos/crash`: сырой счётчик обнуляется, а график `rate` почти не заметил рестарта. Это лучший довод, почему счётчики читают только через `rate`.

13. Алерты

```yaml
      - alert: ApiHighErrorRate
        expr: <доля 5xx> > 0.05
        for: 30s                # условие должно держаться 30 с, а не мигнуть
        labels: { severity: warning }
        annotations: { summary: "Доля 5xx выше 5%" }
```

Правило алерта — это запрос PromQL, который Prometheus вычисляет раз в `evaluation_interval`. Если запрос вернул хотя бы один ряд, условие истинно. `for` требует, чтобы оно держалось заданное время. **Лейблы** (`severity`) нужны для маршрутизации: `critical` — звонок дежурному, `warning` — сообщение в чат. **Аннотации** — текст для человека; `{{ $labels.name }}` подставляет лейбл из результата.

Состояния: **inactive → pending** (условие истинно, ждём `for`) **→ firing**. Если условие пропало в pending, алерт возвращается в inactive, так и не сработав.

У `for` есть цена. Он задаёт минимальную длительность проблемы, которую вы готовы считать инцидентом, и всё, что короче, алерт пропустит по определению. Слишком короткий `for` превращает каждый всплеск в уведомление, слишком длинный откладывает реакцию на настоящий сбой. Короткие, но важные события, такие как рестарт контейнера, ловят не длительностью условия, а отдельным правилом, которое смотрит на сам факт события за окно.

Время от начала сбоя до firing складывается из нескольких слагаемых: до 5 с на очередной scrape, время, пока `rate` за минуту «наберёт» достаточно плохих точек, до 5 с до очередного вычисления правил и `for`. Поэтому даже на стенде алерт срабатывает не мгновенно, а в проде задержка в пару минут нормальна.

Слагаемое про окно легко недооценить. Пусть нагрузка идёт ровно, и вы начинаете слать одну ошибку в секунду. В первые секунды в минутном окне ещё почти одни успешные запросы, и доля 5xx растёт постепенно, пока ошибки не займут заметную часть окна. Только когда она перевалит за 5%, правило станет истинным и запустит отсчёт `for`. Та же инерция работает в обратную сторону: после остановки ошибок алерт гаснет не сразу, а когда плохие точки уйдут из окна.

Правила стенда (`kit/monitoring/alerts.yml`): `ApiDown`, `ApiHighErrorRate`, `ApiSlowRequests`, `ContainerHighMemory`, `ContainerRestarted`.

Алерт	Суть выражения	`for`	severity
`ApiDown`	`up{job="demo-api"} == 0`	30s	critical

`ApiHighErrorRate`	доля 5xx за минуту > 0.05	30s	warning
`ApiSlowRequests`	p95 по всему сервису > 1 с	30s	warning
`ContainerHighMemory`	working set / лимит > 0.8	15s	warning
`ContainerRestarted`	время старта менялось за 5 минут	—	warning

Два выражения заслуживают разбора. В `ContainerHighMemory` знаменатель записан как `(container_spec_memory_limit_bytes{name="demo-api-1"} > 0)`: сравнение в PromQL работает как фильтр и отбрасывает ряд с нулём. Без него контейнер без лимита дал бы деление на ноль и бесконечность, то есть вечный алерт.

```
changes((max by (name) (container_start_time_seconds{name="demo-api-1"}))[5m:5s]) > 0
```

После рестарта cAdvisor заводит новый ряд, и `changes` по одному ряду ничего бы не увидел. Поэтому сначала `max by (name)` сводит ряды в один по имени контейнера, затем **подзапрос** `[5m:5s]` вычисляет это выражение за последние 5 минут с шагом 5 с, и `changes` считает смены значения. `for` здесь не нужен: рестарт — уже свершившийся факт. Зато алерт держится в firing, пока рестарт не уйдёт из пятиминутного окна.

Доставка уведомлений (Telegram, почта, Slack) — Alertmanager или Grafana Alerting. Prometheus только вычисляет правила и отправляет сработавшие алерты в Alertmanager. Тот группирует похожие алерты в одно сообщение, убирает дубли, по лейблам выбирает получателя и умеет временно заглушать (silence) алерты на время плановых работ. В курсе смотрим алерты в UI Prometheus.

14. Какой алерт хороший

- **Алертить по симптомам**, которые видит пользователь: ошибки, латентность, недоступность. Не по «CPU 70%»: высокий CPU при быстрых ответах — это хорошо использованное железо, а не инцидент.
- **Каждый алерт требует действия.** Если на него не нужно реагировать — это шум, удалите или переведите в дашборд.
- **`for`** отсекает кратковременные всплески.
- **Причинные метрики** (память, рестарты) держите как `warning`: они предупреждают о симптоме, который вот-вот появится, но не будят ночью.

Уровень `severity` — это обещание о срочности. `critical` означает «чинить сейчас, даже ночью», и таких алертов должно быть мало: недоступность и массовые ошибки. `warning` означает «посмотреть в рабочее время». Если `warning` по ночам регулярно приходится разбирать срочно, его уровень выбран неверно, и наоборот: `critical`, который неделями никто не трогает, критическим не является.

Наши правила следуют этому: три алерта из пяти — симптомы (`ApiDown`, `ApiHighErrorRate`, `ApiSlowRequests`), два — ресурсные предупреждения.

Alert fatigue. Типичная история: в команде есть алерт «CPU выше 80%», который каждую ночь срабатывает на плановой пересборке отчётов и сам проходит. Через месяц канал с алертами никто не читает, а однажды ночью в том же потоке теряется настоящий `ApiDown`. Лечение одно: каждый алерт, на который ответили «ничего не делали», чинят — поднимают порог, увеличивают `for` или удаляют.

Runbook — короткая инструкция «что делать», привязанная к алерту; обычно ссылку кладут в аннотацию `runbook_url`. Для `ApiSlowRequests` на нашем стенде:

1. Дашборд: когда начался рост p95, совпадает ли он с изменением RPS.
2. Запрос p95 `by (le, controller, action)` — какое действие тормозит.
3. `irm http://localhost:8080/` → `version`: был ли деплой перед началом роста.
4. Совпало с деплоем — откат на предыдущий `sha-` тег, разбор потом.
5. Не совпало — CPU контейнеров (не упирается ли БД), логи `demo-api-1` в Dozzle.

Вторая типичная история — про среднее. Дашборд показывает среднее время ответа около 100 мс, все довольны, а поддержка получает жалобы на «вечную загрузку». Разбивка p99 по действиям находит одно редкое действие, которое выполняется по 5 секунд; в среднем по сервису оно почти невидимо. Именно поэтому алерт стоит на p95, а запрос по `controller, action` держат под рукой.

15. Мониторинг замыкает цикл DevOps

CI/CD из темы 5 проверяет, что код собирается и тесты зелёные. Но тесты не ловят всё: запрос может стать

медленным при корректном результате. Мониторинг — последняя проверка, которая смотрит на реальный сервис после деплоя, и первый шаг к откату.

```
flowchart LR
  M["merge в main"] --> CD["CI/CD:<br/>тесты, образ, деплой"]
  CD --> S["сервис sha-новый"]
  S --> P["Prometheus:<br/>метрики, правила"]
  P -->|"p95 > 1 с, for 30s"| A["алерт firing"]
  A --> D["дашборд + логи:<br/>действие, версия"]
  D --> R["откат на sha-старый"]
  R --> F["фикс через PR"]
  F --> M
```

Чтобы цикл работал, связка «метрики — версия» должна быть однозначной: в `GET /` сервис отдаёт `version`, а тег образа совпадает с коммитом. Тогда вопрос «что изменилось» сводится к сравнению времени роста на графике и времени деплоя. Итоговая лаба 8 проходит этот цикл целиком и меряет два числа: время обнаружения (от merge до firing) и время восстановления (от firing до снятия алерта).

```
sequenceDiagram
  participant API as DemoApi
  participant P as Prometheus
  participant U as Инженер
  loop каждые 5 с
    P->>API: GET /metrics
    API-->>P: счётчики и корзины
  end
  Note over P: правило истинно → pending
  Note over P: истинно 30 с → firing
  P-->>U: алерт в UI (в проде — через Alertmanager)
  U->>P: PromQL по controller, action
```

> **Демонстрация для преподавателя.** При работающей нагрузке откройте Prometheus *Alerts* и запустите `1..40 | ForEach-Object { curl.exe -s -o NUL "$api/api/chaos/slow?ms=1500" }`. Засеките вслух время перехода `ApiSlowRequests` в pending и в firing, затем покажите на дашборде рост p95 и запрос p95 по действиям: виновник — `Chaos/Slow`. Остановите цикл и дождитесь возврата в inactive — это тот же сценарий, что студенты пройдут в лабе 8 с настоящим плохим релизом.

Типичные ошибки

- **`rate()` по gauge** или счётчик без `rate()`: график «лесенкой».
- **Высокая кардинальность лейблов:** id, email, полный путь в лейбле.
- **`le` потерян в `sum by`:** `histogram_quantile` возвращает пустоту.
- **Окно `rate` короче нескольких интервалов scrape:** рваный график и пропуски.
- **Среднее вместо перцентиля** на дашборде и в алертах.
- **Алерты без `for`:** шум, к ним привыкают и перестают читать.
- **Алерты по причинам** (CPU, память) с уровнем critical вместо алертов по симптомам.
- **Дашборды «накликаны» руками** и не лежат в git.
- **Мониторинг в том же compose-проекте, что и приложение:** `down` приложения гасит и мониторинг.

Вопросы для самопроверки

1. Почему для счётчика нужен `rate()`?
2. Почему алертим на p95, а не на среднее время ответа?
3. Чем `pending` отличается от `firing` и зачем `for`?
4. Какие метрики даёт сервис, а какие — cAdvisor? Что увидит только cAdvisor?
5. Рост 409 на `ChangeStatus` — это проблема сервиса или клиентов? Как проверить?
6. Что покажет `up{job="demo-api"}`, если сервис жив, но на него нет запросов? А если контейнер остановлен?
7. Почему id задачи нельзя сделать лейблом метрики, а `endpoint="api/tasks/{id:int}"` можно?
8. Как по гистограмме посчитать долю запросов быстрее 512 мс? Почему не 500 мс?

Практика: лаба 7

Подключите мониторинг к сервису задач, найдёте на графиках свои контроллеры и методы, добавите панель на дашборд и увидите, как срабатывают алерты на сбоях из лабы 6. В лабе 8 этот мониторинг поймает плохой релиз.