

Генерация документации и runbooks для инцидентов в SRE с использованием мультимодальной модели GPT-5.2

Саяпин Никита Анатольевич

Приамурский государственный университет им.Шолом-Алейхема

Студент

Аннотация

В статье рассматривается процесс автоматической генерации runbooks (руководств по реагированию на инциденты) и связанной документации в практике Site Reliability Engineering (SRE) на основе текстового описания сервиса и типичного инцидента с применением мультимодальной нейросети GPT-5.2. Показывается, как модель структурирует процедуры, включает шаги диагностики, устранения и пост-мортем анализа, а также оценивается влияние уровня детализации промпта на качество, полноту и практическую применимость получаемых документов.

Ключевые слова: GPT-5.2, Site Reliability Engineering, runbooks, реагирование на инциденты, автоматизация SRE, генерация документации, incident management.

Generation of incident runbooks and documentation in SRE using the GPT-5.2 multimodal model

Sayapin Nikita Anatolyevich

Sholom-Aleichem Priamursky State University

Student

Abstract

The article explores the automatic generation of runbooks (incident response guides) and related documentation in Site Reliability Engineering (SRE) based on a text description of the service and typical incident using the GPT-5.2 multimodal neural network. It demonstrates how the model structures procedures, includes diagnostic and remediation steps, and post-mortem analysis, while evaluating the impact of prompt detail level on the quality, completeness, and practical applicability of the generated documents.

Keywords: GPT-5.2, Site Reliability Engineering, runbooks, incident response, SRE automation, documentation generation, incident management.

1. Введение

1.1 Актуальность

В практике Site Reliability Engineering (SRE) значительная часть рабочего времени инженеров уходит на реагирование на инциденты, их анализ и документирование. Runbooks — это структурированные руководства,

описывающие последовательность действий при типичных сбоях (outage, high latency, OOMKilled поды и т.д.), — помогают снизить время восстановления (MTTR), уменьшить нагрузку на on-call инженеров и обеспечить единообразие реакции команды. Однако создание и поддержание runbooks в актуальном состоянии требует значительных усилий: нужно описывать симптомы, шаги диагностики, команды для исправления, критерии эскалации, контакты и шаблоны пост-мортем отчётов.

Мультимодальные модели нового поколения, такие как GPT-5.2, позволяют автоматизировать этот процесс. Модель способна по краткому описанию сервиса и характера инцидента генерировать полноценный runbook, включая логическую структуру, конкретные команды, лучшие практики SRE (blameless culture, SLO/SLI) и даже предложения по автоматизации (playbooks в формате YAML или скрипты). Такой подход особенно актуален в условиях роста сложности распределённых систем, микросервисов и облачных инфраструктур, где количество потенциальных инцидентов растёт экспоненциально, а команды SRE часто ограничены в ресурсах.

1.2 Обзор исследований

В статье А. Р. Очередыко рассматриваются особенности и функции систем реагирования на инциденты информационной безопасности. Представлен анализ современных решений IRP и описан процесс реагирования на типовые инциденты в системах этого класса. На основании экспертных мнений сформирован перечень критериев, которые были распределены в группы по зонам функциональной ответственности для дальнейшего сравнения работы IRP-систем [1]. В статье А.В. Калачинского представлена технология и методика генерации документации в формате PDF, которая описывает работу алгоритмов, разработанных на основе автоматного подхода к проектированию ПО. Рассмотрен пример преобразования графа переходов автомата в документ, содержащий описание его реализации [2]. Д. Ю. Петров рассматривает вопросы создания структуры информационной системы для разработки цифрового двойника непрерывного технологического процесса. На основе анализа применения цифровых двойников производства предложены этапы создания цифрового двойника непрерывного производства с учетом требований действующих отечественных стандартов области разработки автоматизированных систем управления непрерывными технологическими процессами [3]. В. В. Денисенко описал систему управления инцидентами в соответствии с требованиями государственных регуляторов по статусам субъектов регулирования. Рассмотрены исключительные случаи присвоения одному субъекту нескольких статусов в направлении отчетности перед регуляторами [4]. А.С. Белоногов исследовал взаимодействие открытой системы IRP, разрабатываемой международным сообществом, с межсетевыми экранами российского производства. На основе функций системы «TheHive» реализован функционал по блокированию атак с внешнего периметра. Представлен алгоритм модуля взаимодействия с межсетевыми экранами для предотвращения компьютерных атак, для

разработки которого использован язык Python, в частности, модуль textFSM [5].

1.3 Цель исследования

Целью исследования является демонстрация возможностей мультимодальной нейросети GPT-5.2 в генерации runbooks и сопутствующей SRE-документации на основе текстового описания сервиса и инцидента, а также оценка качества получаемых документов с точки зрения структуры, полноты, соответствия лучшим практикам SRE и зависимости результата от детализации входного промпта.

2 Материалы и методы

В исследовании применяется мультимодальная модель GPT-5.2, которая воспринимает текстовые описания сервиса, архитектуры и характера инцидента, а затем генерирует структурированные документы в формате Markdown или простого текста. Модель рассматривается как инструмент преобразования естественного языка в готовые руководства без предварительной подготовки шаблонов.

Подготавливаются промпты различной степени детализации: от кратких описаний проблемы (симптомы и сервис) до средндетализированных (с указанием стека технологий, SLO и желаемой структуры) и высоко детализированных (с полным описанием архитектуры, метрик, прошлых инцидентов и требований к пост-мортем).

Выбираются типичные SRE-сценарии: падение подов в Kubernetes из-за нехватки памяти, высокий процент 5xx ошибок на веб-сервисе, деградация базы данных, потеря кворума в кластере и аналогичные ситуации. Генерируемые runbooks анализируются по критериям: логическая структура (разделы: симптомы, диагностика, remediation, verification, post-mortem), наличие конкретных команд и инструментов (kubectl, promql, logs), соответствие принципам blameless culture и SRE (Google SRE practices), читаемость и практическая применимость. Особое внимание уделяется тому, как детализация промпта влияет на глубину и точность документа.

Оценка осуществляется путём визуального анализа структуры и содержания сгенерированных текстов.

3. Результаты

3.1 Генерация runbook по минимальному описанию

На первом этапе исследовалась способность модели создавать полноценный runbook на основе крайне короткого промпта, где указана только суть инцидента без упоминания стека технологий, желаемой структуры или дополнительных требований. Это имитирует реальную ситуацию, когда on-call инженер в 3 часа ночи быстро формулирует задачу и ожидает от ИИ готового руководства.

Промпт:

«Создай runbook для инцидента: поды в Kubernetes падают с OOMKilled.»

GPT-5.2 самостоятельно интерпретировала запрос и сгенерировала детализированный, хорошо структурированный документ в формате Markdown с чёткими разделами, таблицами и практическими рекомендациями. Модель начала с краткого описания проблемы и целей runbook, затем перешла к многоуровневому плану действий.

Описание проблемы (как указано моделью):

Контейнеры в поде убиваются kubelet'ом из-за превышения лимита памяти (memory limit). В `kubectl describe pod` видно: Reason: OOMKilled, Exit Code: 137.

Цели runbook:

Быстро стабилизировать ситуацию

Найти виновника

Предотвратить повторение в ближайшие часы/дни

Шаг 0. Быстрая стабилизация (5–10 минут)

Модель предложила последовательность из пяти действий с командами, ожидаемыми результатами и временными рамками:

1) Найти проблемные поды: `kubectl get pods --all-namespaces -o wide --sort-by='.status.containerStatuses[0].restartCount'` → поды с большим `restartCount` и статусом `OOMKilled`.

2) Посмотреть последние павшие поды: `kubectl get pods -A | grep -i oom` или через события.

3) Посмотреть события по поду: `kubectl get events --field-selector involvedObject.name=<pod-name> --sort-by=.metadata.creationTimestamp` → поиск Reason: Evicted / Killing / OOMKilled.

4) Временно увеличить лимит: отредактировать `Deployment/StatefulSet` → `memory.limit × 1.5–2×` → под перестал рестартоваться.

5) Если много подов — временно отключить HPA: `kubectl scale hpa <name> --replicas=0` → потом обратно → снижение нагрузки.

Шаг 1. Диагностика — где именно проблема (10–20 минут)

Здесь модель создала таблицу из 8 пунктов с проверками по порядку:

1) Лимиты и requests: `kubectl get pod <pod-name> -o jsonpath='{.spec.containers[*].resources}'` → плохо, если `limit < 1.5–2× request` или `limit` не стоит.

2) Реальное потребление: `kubectl top pod -A | sort -rnk 3` → >90–95% от лимита стабильно.

3) Пиковое потребление: `kubectl describe pod` → Limits и Last State → OOMKilled.

4) Логи перед смертью: `kubectl logs <pod-name> --previous` → часто Java: GC-логи, `OutOfMemoryError`.

5) Memory usage в Prometheus: `container_memory_working_set_bytes{namespace="...",pod=~"$pod"} / 1024^3` → график упирается в лимит и резко падает («пила»).

6) Node pressure: `kubectl describe node <node-name>` → `MemoryPressure: True`.

7) cgroup на нодe: `ssh на ноду` → `cat /sys/fs/cgroup/memory/memory.usage_in_bytes` → подходит к лимиту ноды.

8) Java-метрики: `jcmd`, `jmap -histo`, `JMX` → Old Gen заполнен, Full GC каждые 5–30 сек.

Шаг 2. Наиболее вероятные причины и быстрые действия

Модель представила таблицу из 6 причин с симптомами, временными и долгосрочными решениями:

Слишком низкий `memory limit` → поднять $\times 1.5-2\times$ → VPA, пересчёт лимитов.

Memory leak → рестарт (временно) → профилирование (`heap dump`, `async-profiler`).

Неправильная JVM heap → `-XX:MaxRAMPercentage=70-75` → Container aware JVM (21+).

Sidecar жрёт память → отключить временно → обновить/уменьшить лимит sidecar.

Node исчерпал память → `cordon/drain` → увеличить пул, Cluster Autoscaler.

HugePages leak → отключить `hugepages`.

Шаг 3. Долгосрочные меры предотвращения

Модель завершила рекомендациями: внедрить VPA (рекомендуемый режим), настроить алерты в Prometheus (`container_memory_working_set_bytes / container_spec_memory_limit_bytes > 0.85` → warning, `> 0.95` → critical, `OOMKilled > 0` → P1/P2), добавить memory throttling, для Java — флаги `-XX:+UseContainerSupport`, `-XX:MaxRAMPercentage=75`, `-XX:+HeapDumpOnOutOfMemoryError`.

Интересно, что при минимальном промпте модель не только создала базовый чек-лист, но и добавила таблицы, временные рамки, JVM-специфику, алерты и профилактику — это делает `runbook` гораздо полезнее, чем ожидалось от простого запроса. Документ готов к копированию в Confluence или GitHub wiki (Рис. 1)

Шаг 0. Быстрая стабилизация (5–10 минут)		
Действие	Команда / Действие	Ожидаемый результат
1. Найти проблемные поды	<code>kubectl get pods --all-namespaces -o wide --sort-by='.status.containerStatuses[0].restartCount'</code>	Поды с большим restartCount и статусом OOMKilled
2. Посмотреть последние 5–10 павших подов	<code>kubectl get pods -A grep -i oom</code> или через событие	—
3. Посмотреть события по проблемному поду	<code>kubectl get events --field-selector involvedObject.name=<pod-name> --sort-by=.metadata.creationTimestamp</code>	Ищем Reason: Evicted / Killing / OOMKilled
4. Временно увеличить лимит (если возможно)	Отредактировать Deployment/StatefulSet → <code>memory.limit × 1.5–2×</code>	Под перестал рестартоваться
5. Если много подов — включить HPA (если есть)	<code>kubectl scale hpa <name> --replicas=0</code> → потом обратно	Снижение нагрузки на проблемные поды

Рисунок 1 — Фрагмент runbook: Шаг 0 (стабилизация) и таблица причин/решений

3.2 Runbook средней детализации с фокусом на метриках и визуализации

На втором этапе использовался более конкретный промпт, где указан стек (Kubernetes), ключевые инструменты диагностики (Prometheus и Grafana), необходимость команд kubectl, перечня причин и шаблона пост-мортем. Это позволяет оценить, насколько модель справляется с задачей, когда есть ориентиры, но нет полного технического задания.

Промпт: «Напиши runbook для SRE: высокий процент 5xx на веб-сервисе в Kubernetes. Укажи шаги диагностики через Prometheus и Grafana, команды kubectl, возможные причины и пост-мортем шаблон.»

GPT-5.2 сгенерировала современный, подробный и структурированный runbook, ориентированный на реальные SRE-практики (Google SRE, Kubernetes docs). Документ включает название инцидента, severity, описание, цели, многоуровневые шаги с таблицами, конкретные метрики и полноценный шаблон пост-мортем в стиле blameless culture.

Название инцидента: Высокий процент 5xx-ошибок на веб-сервисе (e.g., >5% от запросов) Severity: P1–P0 (в зависимости от трафика и бизнес-влияния; P0 если >10–20% и влияет на пользователей) Описание: Веб-сервис возвращает 5xx-ошибки (500 Internal Server Error, 502 Bad Gateway, 503 Service Unavailable и т.д.) в значительном объёме. Это видно в логировании, метриках

(например, через Ingress) или мониторинге (Prometheus/Grafana). Симптомы: пользователи жалуются на недоступность, алерты на SLO violation (error rate > threshold).

Цели runbook:

- Стабилизировать сервис (вернуть error rate < 1–2%)
- Диагностировать root cause
- Предотвратить рецидивы
- Подготовить пост-мортем для анализа

Шаг 0. Быстрая стабилизация (5–15 минут)

Модель сразу предложила действия по снижению влияния на пользователей:

- 1) Проверить текущий error rate в Grafana: дашборд "HTTP Errors" → метрика $\frac{\text{sum}(\text{rate}(\text{http_server_errors_total}\{\text{code}=\sim"5.."\}[5\text{m}]))}{\text{sum}(\text{rate}(\text{http_requests_total}[5\text{m}]))}$ → подтверждение > threshold (e.g., 5%).
- 2) Откатить недавний деплой: `kubectl rollout undo deployment/<name> -n <ns>` или через CI/CD (ArgoCD/Flux) → возврат к стабильной версии.
- 3) Масштабировать поды: `kubectl scale deployment <name> --replicas=<N+20%>` → снижение нагрузки.
- 4) Включить circuit breaker (если Istio/Linkerd): отредактировать VirtualService → добавить retries, timeouts.
- 5) Редирект трафика: обновить DNS/Route53 или Traffic Manager на резервный кластер.

Шаг 1. Диагностика (15–30 минут) Модель создала таблицу из 8 проверок с инструментами и критериями «плохо»:

- 1) Общий error rate: Grafana панель "HTTP Response Codes" → `histogram_quantile(0.95, sum(rate(http_request_duration_seconds_bucket{code=~"5.."}[5m])) by (le))` → P95 latency > 500ms, 5xx > 5%.
- 2) Error rate по endpoint'ам: Prometheus → `sum(rate(http_server_errors_total{code=~"5..", path=~"/api/.*"}[5m])) by (path)` → конкретный endpoint > 10%.
- 3) Поды с ошибками: `kubectl get pods -n <ns> -o wide | grep -i error` или `kubectl top pods` → CrashLoopBackOff или высокий CPU/Mem.
- 4) Логи подов: `kubectl logs -n <ns> <pod-name> --tail=100 | grep -i "error|exception|500"` → стек-трейсы, DB timeouts.
- 5) Метрики нод: Grafana "Node Exporter" → $\frac{\text{node_memory_MemAvailable_bytes}}{\text{node_memory_MemTotal_bytes}} < 0.2$ → ноды с низкой памятью.
- 6) Ingress/Proxy errors: `sum(rate(nginx_ingress_controller_requests{status=~"5.."}[5m])) by (ingress)` → 502/504 от Ingress.
- 7) Зависимости: Grafana "Service Dependencies" → latency к бэкендам (e.g., Istio метрики).

8) Трафик spikes: `sum(rate(http_requests_total[5m]))` → резкий рост запросов.

Дополнительно модель добавила команды:

`kubectl describe pod <pod-name>` — события, conditions.

`kubectl exec -it <pod-name> -- curl localhost:8080/health` — health check внутри пода.

`kubectl get events -n <ns> --sort-by=.metadata.creationTimestamp` — корреляция событий.

Шаг 2. Наиболее вероятные причины и быстрые действия

Таблица из 7 причин с симптомами, временными и долгосрочными решениями:

1) Перегрузка ресурсов → высокий CPU, OOM → масштабировать → HPA/VPA.

2) Проблемы с зависимостями (DB down) → таймауты → failover → circuit breakers.

3) Баг в коде → ошибки на endpoint'ах → откат → code review, canary.

4) Ingress misconfig → 502/504 → увеличить timeouts → обновить controller.

5) DDoS/трафик spike → пост requests → WAF, rate limiting → autoscaling, CDN.

6) Health checks failing → 503 от LB → исправить probes → тестировать в CI.

7) Network issues → intermittent 5xx → restart kube-proxy → upgrade CNI.

Шаг 3. Долгосрочные меры предотвращения

Модель завершила рекомендациями:

Алерты: `sum(rate(http_server_errors_total{code=~"5.."}[5m])) / sum(rate(http_requests_total[5m])) > 0.05` → P1; Latency P99 >500ms → warning; SLO burn rate.

Observability: Tracing (Jaeger/OpenTelemetry).

Chaos engineering: Chaos Mesh.

Обновить дашборды Grafana.

Graceful degradation в приложении.

Шаблон пост-мортем Полноценный шаблон в стиле Google SRE: Summary, Impact, Trigger, Root Cause, Timeline, Detection, Response, Recovery, Lessons Learned, Action Items (с owner и deadline), следующий review.

Модель создала runbook, который можно сразу использовать в команде: он содержит конкретные PromQL-запросы, kubectl-команды, таблицы, алерты и blameless пост-мортем. Всё это при промпте средней детализации — результат гораздо глубже и профессиональнее, чем от минимального запроса (Рис. 2).

Что			Что
#	проверяем	Инструмент / Команда	означает «плохо»
1	Общий error rate по сервису	Grafana: панель "HTTP Response Codes" → <code>histogram_quantile(0.95, sum(rate(http_request_duration_seconds_bucket{code=~"5.."}[5m]))) by (le))</code>	P95 latency >500ms, 5xx >5%
2	Error rate по endpoint'ам	Prometheus query: <code>sum(rate(http_server_errors_total{code=~"5..", path=~"/api/.+"}[5m])) by (path)</code>	Конкретный endpoint (e.g., /checkout) >10% ошибок
3	Поды с ошибками (kubectl)	<code>kubectl get pods -n <ns> -o wide grep -i error</code> или <code>kubectl top pods -n <ns> --sort-by cpu</code>	Поды в CrashLoopBackOff или с высоким CPU/Mem
4	Логи подов перед ошибками	<code>kubectl logs -n <ns> <pod-name> --tail=100 grep -i "error exception 500"</code>	Стек-трейсы, DB timeouts, OOM
5	Метрики нод (node pressure)	Grafana: "Node Exporter" дашборд → <code>node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes < 0.2</code>	Ноды с низкой памятью/CPU, eviction подов
6	Ingress/Proxy errors	Prometheus: <code>sum(rate(nginx_ingress_controller_requests{status=~"5.."}[5m])) by (ingress)</code>	502/504 от Ingress (e.g., upstream timeouts)

Рисунок 2 — Фрагмент runbook: таблица диагностики Prometheus/Grafana и шаблон пост-мортем

3.3 Генерация сложного runbook по высоко детализированному описанию

На третьем этапе применялся максимально подробный промпт с указанием конкретной архитектуры (Patroni + HAProxy + Kubernetes), необходимых элементов диагностики (pg_stat_statements, EXPLAIN, pgBadger), методов устранения (scale replicas, tune config, vacuum), разделов verification, автоматизации (alertmanager rules), пост-мортем шаблона и контактов. Это позволяет модели создать runbook на уровне enterprise, с акцентом на практическую применимость в сложных распределённых системах.

Промпт: «Генерируй подробный SRE runbook: деградация PostgreSQL (high query latency). Архитектура: Patroni + HAProxy + Kubernetes. Включи: симптомы, диагностика (pg_stat_statements, EXPLAIN, pgBadger), remediation (scale replicas, tune config, vacuum), verification, автоматизация (alertmanager rules), post-mortem template, contacts.»

GPT-5.2 сгенерировала современный, всесторонний SRE runbook, ориентированный на высокую задержку запросов в PostgreSQL. Документ включает название инцидента, severity, описание, контакты, цели, симптомы, шаги (стабилизация, диагностика, remediation, verification, автоматизация, долгосрочные меры) и шаблон пост-мортем, с таблицами, командами, SQL-запросами, YAML-кодом и практическими рекомендациями.

Название инцидента: Высокая задержка запросов в PostgreSQL (query latency > threshold, e.g., P95 > 500ms) Severity: P1–P0 (в зависимости от влияния: P0 если >50% запросов затронуты, влияет на бизнес-SLO; P1 если частичная деградация) Описание: PostgreSQL кластер (управляемый Patroni для HA) в Kubernetes показывает повышенную задержку запросов. Трафик балансируется через HAProxy. Симптомы включают медленные ответы от приложений, таймауты в сервисах, алерты на latency. Это может привести к cascading failures в upstream-сервисах. Архитектура: Patroni (leader election, failover), HAProxy (load balancing на replicas), Kubernetes (StatefulSet для PG pods).

Контакты:

- 1) On-call SRE: PagerDuty ротация "SRE-DB" (escalate если >15 мин без прогресса)
- 2) DBA Team: Slack #dba-support (primary experts для tuning/vacuum)
- 3) App Owners: Slack #app-team (для координации с upstream-приложениями)
- 4) Infra Team: Если K8s/node issues – Slack #infra-k8s
- 5) Escalation: Если P0 – notify CTO via PagerDuty
- 6) Цели runbook:
- 7) Стабилизировать кластер (снизить latency < threshold)
- 8) Диагностировать root cause
- 9) Восстановить и предотвратить рецидивы
- 10) Автоматизировать детекцию/реакцию

Симптомы:

- 11) Высокая P95/P99 latency запросов (>500ms–1s, в зависимости от SLO)
- 12) Алерты: "PG_Query_Latency_High" или "App_Response_Time_Degraded"
- 13) Пользовательские жалобы: медленный UI, timeouts в API
- 14) Метрики: высокий pg_stat_statements.mean_exec_time, CPU/IO wait на PG pods
- 15) Логи: ошибки в Patroni (failover events), HAProxy (backend timeouts)
- 16) Kubernetes events: Pod restarts, OOM в PG pods
- 17) Шаг 0. Быстрая стабилизация (5–15 минут): Цель: Снизить влияние, пока диагностика идет.
- 18) Проверить текущую latency: Grafana дашборд "PG Performance" → histogram_quantile(0.95,

`sum(rate(pg_stat_statements_mean_exec_time_bucket[5m])) by (le)) → подтверждение деградации (latency > threshold).`

19) Балансировать трафик: Проверить HAProxy `config kubectl exec -n <ns> <haproxy-pod> -- haproxy -f /etc/haproxy/haproxy.cfg -c → добавить timeout server 30s → снижение timeouts от HAProxy.`

20) Временно scale read replicas: `kubectl scale statefulset <pg-replica> -- replicas=<N+1> (Patroni auto-configures) → разгрузка leader, read-only queries faster.`

21) Rate limit upstream traffic: В Istio/Linkerd добавить VirtualService с rate limits или circuit breakers → защита от overload.

22) Failover если leader overloaded: `Patroni patronictl failover (manual, если auto-failover не сработал) → новый leader с меньшей нагрузкой.`

Шаг 1. Диагностика (15–45 минут): Используйте `pg_stat_statements` для топ-запросов, EXPLAIN для анализа планов, pgBadger для лог-анализа. Начните с высокоуровневого, углубляйтесь. Таблица из 8 проверок:

- Топ-запросы по latency: В PG shell `SELECT query, mean_exec_time, calls FROM pg_stat_statements ORDER BY mean_exec_time DESC LIMIT 10; → high % total time.`
- План выполнения запроса: `EXPLAIN (ANALYZE, BUFFERS) <slow_query>; → seq scan вместо index, high % time on I/O.`
- Лог-анализ: `pgBadger /path/to/postgresql.log → high % I/O, locks.`
- PG метрики (Prometheus): `sum(rate(pg_stat_database_xact_commit[5m]))` или `pg_settings_work_mem → high commit rate, spills to disk.`
- Patroni status: `patronictl list` или `kubectl get pods -l cluster-name=<pg-cluster> → leader flapping, replicas lagging (>1min).`
- HAProxy backend stats: `kubectl exec <haproxy-pod> -- echo "show stat" | socat /var/run/haproxy.sock stdio → high % errors on backends.`
- K8s resources: `kubectl top pods -n <ns> | grep pg → pods >80% CPU, IO wait high.`
- Вакуум/bloat stats: `SELECT relname, pg_table_size(relid) FROM pg_stat_user_tables ORDER BY pg_table_size DESC; → tables с bloat >50%, last_vacuum старше 1 дня.`

Дополнительно: Войдите в PG pod `kubectl exec -it <pg-pod> -- psql -U <user> <db>;` включите `pg_stat_statements` если не: `ALTER SYSTEM SET shared_preload_libraries = 'pg_stat_statements';`

Шаг 2. Remediation (30–60 минут): Таблица из 6 причин:

- 1) Перегрузка (high load) → high CPU/IO → scale replicas, rate limit в HAProxy → HPA для PG StatefulSet, query optimization.
- 2) Плохой query plan → seq scans → ADD INDEX → регулярный ANALYZE, pg_repack для реорганизации.
- 3) Config tuning needed → low work_mem → `ALTER SYSTEM SET work_mem = '16MB'; SELECT pg_reload_conf(); → авто-tuning (pg_autotune).`

- 4) Bloat / vacuum issues → high % table size → VACUUM FULL; или AUTOVACUUM on → настроить autovacuum_scale_factor=0.1.
- 5) Patroni/HA issues → leader overload → manual failover → upgrade Patroni, добавить больше replicas.
- 6) K8s/node problems → high node CPU → cordon/drain node → Cluster Autoscaler, affinity rules.

Шаг 3. Verification: После remediation:

- Мониторьте latency 10–15 мин: Grafana панель "PG Latency" – должно быть < threshold.
- Запустите тестовые запросы: EXPLAIN ANALYZE <query>; – проверить улучшение плана.
- Проверьте pg_stat_statements: mean_exec_time уменьшилось.
- HAProxy stats: qtime <100ms, no errors.
- Patroni: все replicas healthy, no lag. Если OK – declare resolved, мониторьте 1 час.

Шаг 4. Автоматизация: Настройте алерты в Prometheus/Alertmanager. Примеры Alertmanager rules (в prometheus-rules.yaml):

```
YAML
groups:
- name: postgres-alerts
  rules:
  - alert: PG_Query_Latency_High
    expr: histogram_quantile(0.95,
sum(rate(pg_stat_statements_mean_exec_time_bucket[5m])) by (le)) > 500 # ms
    for: 5m
    labels:
      severity: critical
    annotations:
      summary: "High PG query latency (instance {{ $labels.instance }})"
      description: "P95 latency > 500ms for 5 min"
  - alert: PG_Bloat_High
    expr: sum(pg_table_size) by (relname) > 10e9 # >10GB bloat
    for: 10m
    labels:
      severity: warning
    annotations:
      summary: "Table bloat detected"
  - alert: Patroni_Leader_Flap
    expr: changes(patroni_leader[5m]) > 2
    for: 1m
    labels:
      severity: critical
    annotations:
      summary: "Patroni leader flapping"
  • Интегрируйте с PagerDuty/Slack для уведомлений.
```

- Авто-remediation: Используйте Operator (Crunchy Postgres Operator) для auto-scale replicas на основе метрик.

Шаг 5. Долгосрочные меры предотвращения:

- Регулярный аудит: Еженедельно запускать pgBadger на logs, проверять pg_stat_statements.
- Tuning: Внедрить pg_tune или auto-config на основе workload.
- Мониторинг: Добавить дашборды для bloat, vacuum stats, query plans.
- Тестирование: Load testing с pgbench в staging, chaos engineering (inject latency в K8s).
- Документация: Обновить runbook после каждого инцидента.

Модель создала runbook, идеально адаптированный под промпт: он интегрирует Patroni, HAProxy, Kubernetes с указанными инструментами диагностики и remediation, добавляя таблицы, YAML-alerters, SQL-команды и blameless пост-мортем. При высокой детализации результат — готовый к использованию документ, который может снизить MTTR в реальной команде SRE (Рис. 3).

Шаг 1. Диагностика (15–45 минут)			
Используйте pg_stat_statements для топ-запросов, EXPLAIN для анализа планов, pgBadger для лог-анализа. Начните с высокоуровневого, углубляйтесь.			
Что # проверяем	Инструмент / Команда	Что означает «плохо»	
1 Топ-запросы по latency	B PG shell: <code>SELECT query, mean_exec_time, calls FROM pg_stat_statements ORDER BY mean_exec_time DESC LIMIT 10;</code>	Несколько запросов >80% total time, high mean_exec_time	
2 План выполнения запроса	<code>EXPLAIN (ANALYZE, BUFFERS) <slow_query>;</code> → искать seq scans, high cost	Seq scan вместо index, sort spills to disk	
3 Лог-анализ	pgBadger: <code>pgbadger /path/to/postgresql.log -o report.html</code> → смотреть "Slow Queries" section	High % time on I/O, locks, или specific queries	
4 PG метрики (Prometheus)	Query: <code>sum(rate(pg_stat_database_xact_commit[5m]))</code> или <code>pg_settings_work_mem</code>	High commit rate, low work_mem → spills	
5 Patroni status	<code>patronictl list</code> или <code>kubectl get pods -l cluster-name=<pg-cluster></code>	Leader flapping, replicas lagging (>1min)	
6 HAProxy backend stats	<code>kubectl exec <haproxy-pod> -- echo "show stat" socat /var/run/haproxy.sock stdio</code>	High qtime (queue time), errors on backends	
7 K8s resources	<code>kubectl top pods -n <ns> grep pg</code> → CPU/Mem usage	Pods >80% CPU, IO wait high	

Рисунок 3 — Фрагмент runbook: таблица диагностики и alerting rules YAML

4 Выводы

Проведённое исследование демонстрирует, что мультимодальная модель GPT-5.2 обладает высокой эффективностью в генерации runbooks и сопутствующей документации для инцидентов в практике Site Reliability Engineering. Модель успешно интерпретирует текстовые описания сервисов, архитектур и характера инцидентов, преобразует их в структурированные руководства с чёткими разделами, конкретными командами, метриками и рекомендациями по автоматизации. При увеличении детализации промпта качество документов значительно повышается: от простых чек-листов с базовыми шагами стабилизации и диагностики до полноценных enterprise-level runbooks, включающих таблицы причин, YAML-код для алертов, SQL-запросы, JVM- или PostgreSQL-специфику, шаблоны пост-мортем и меры предотвращения. Модель часто самостоятельно добавляет полезные элементы, такие как временные рамки действий, критерии "плохо" в диагностике, интеграцию с инструментами вроде Prometheus, Grafana или

Patroni, а также принципы blameless culture, что делает генерируемые материалы не только информативными, но и практически применимыми в реальных SRE-командах.

Использование GPT-5.2 позволяет существенно сократить время на создание и обновление runbooks, снизить нагрузку на on-call инженеров, унифицировать подходы к реагированию на инциденты и способствовать снижению MTTR в сложных системах вроде Kubernetes с PostgreSQL или веб-сервисами. Технология особенно ценна в условиях роста микросервисных архитектур, где количество потенциальных сбоев увеличивается, а ресурсы команд ограничены. Вместе с тем существуют ограничения: при минимальных промптах модель может упускать нюансы конкретной инфраструктуры, такие как кастомные конфигурации или интеграции, поэтому генерируемые документы всегда требуют ручной проверки и адаптации специалистами. Кроме того, модель не имеет доступа к реальному состоянию системы, что делает необходимым тестирование предложенных команд и алертов в staging-среде перед внедрением в production.

В целом технология открывает широкие перспективы для автоматизации SRE-процессов: от быстрого прототипирования runbooks для новых сервисов до интеграции с CI/CD для автоматического обновления документации. В будущем такие модели могут стать основой для полностью автоматизированных систем incident management, где ИИ не только генерирует руководства, но и предлагает proactive меры на основе анализа логов и метрик в реальном времени.

Библиографический список

1. Очерedyко А. Р., Бачманов Д. А., Путятю М. М., Макарян А. С. Исследование IRP-систем на основе анализа механизмов реагирования на инциденты информационной безопасности // Прикаспийский журнал: управление и высокие технологии. 2021. № 1(53). С. 74-82.
2. Калачинский А. В., Яценко И. Н. Генерация описания работы автоматных программ в документ формата PDF // Системы управления и обработки информации. 2019. № 1(44). С. 93-98.
3. Петров Д. Ю. Структура информационной системы для разработки цифрового двойника непрерывного технологического процесса // Математические методы в технологиях и технике. 2021. № 5. С. 60-65.
4. Денисенко В. В., Гончаров А. М., Маслов А. А. Особенности управления инцидентами информационной безопасности // Вестник Воронежского института ФСИИ России. 2024. № 4. С. 56-64.
5. Белоногов А. С., Будник М. Г., Мельников А. В. Инструментальная средареагирования на инциденты информационной безопасности // Вестник УрФО. Безопасность в информационной сфере. 2024. № 4(54). С. 22-29.