

# Vitess Demo

---

Autorzy:

- Szymon Twardosz
- Dominik Pilipczuk
- Krzysztof Tranquilo Dziechciarz
- Wiktor Gut

## 1. Wprowadzenie

Celem projektu jest przedstawienie studium przypadku systemu Vitess – open-sourcowej platformy służącej do skalowania baz danych MySQL w środowiskach chmurowych i rozproszonych. Vitess został stworzony przez YouTube jako odpowiedź na rosnące potrzeby skalowalności i dostępności danych w dużych systemach produkcyjnych, gdzie tradycyjne podejście do baz danych relacyjnych okazywało się niewystarczające. Vitess łączy w sobie zalety tradycyjnych baz danych (jak ACID i SQL) z elastycznością architektur opartych na mikrousługach i kontenerach. Umożliwia m.in. sharding, replikację, przełączanie awaryjne oraz zarządzanie schematem w sposób spójny i zautomatyzowany. Dzięki integracji z Kubernetesem i innymi narzędziami cloud-native, Vitess idealnie wpisuje się w potrzeby nowoczesnych, skalowalnych aplikacji.

Przykładowe firmy i usługi, które korzystają z technologii Vitess, to:

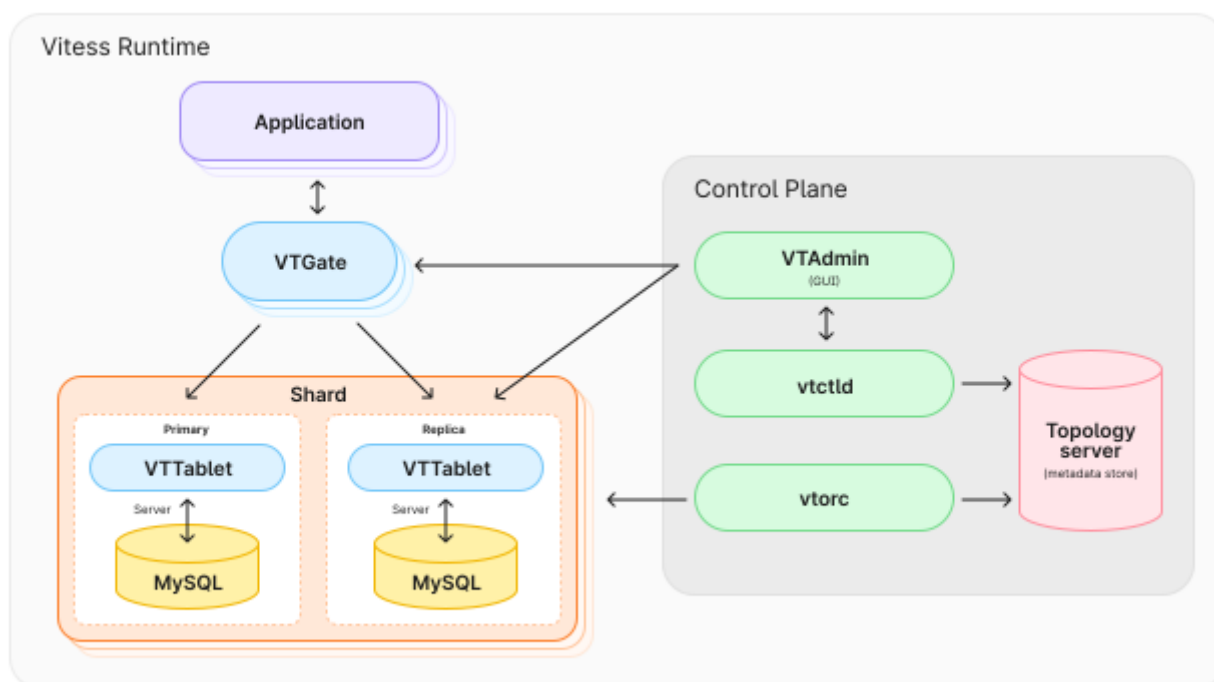
- YouTube – gdzie projekt się narodził, jako rozwiązanie problemów skalowalności bazy danych,
- Slack – dla obsługi ogromnej ilości wiadomości i użytkowników w czasie rzeczywistym,
- GitHub – do obsługi skomplikowanej infrastruktury danych przy zachowaniu wysokiej dostępności i wydajności.

## 2. Wstęp teoretyczny

Vitess składa się z kilku kluczowych komponentów, które współpracują, aby zapewnić wydajność, skalowalność oraz wysoką dostępność baz danych. Oto najważniejsze z nich:

- Vtorc – odpowiada za automatyczne zarządzanie topologią replikacji MySQL oraz wykrywanie i reagowanie na awarie w klastrze bazodanowym.
- VTGate – brama (proxy) łącząca aplikacje z systemem Vitess. Odpowiada za przyjmowanie zapytań SQL od klientów i ich kierowanie do odpowiednich shardów i replik.
- VTablet – serwis działający obok instancji MySQL w każdej replice. Odpowiada między innymi za zarządzanie instancją MySQL oraz odpowiadanie na zapytania przychodzące z VTGate.
- vtctld – interfejs administracyjny służący do zarządzania klastrem Vitess. Umożliwia między innymi monitorowanie stanu klastra czy tworzenie nowych shardów oraz replik.
- Shardy – instancje baz danych, do których kierowane są zapytania. Każdy z nich posiada własną instancję VTablet, która jest pośrednikiem w komunikacji między instancją bazy danych a VTGate.

Architektura tej technologii wygląda następująco (obrazek wzięty z oficjalnej dokumentacji Vitess).



### 3. Opis koncepcji

W ramach początkowej wizji projektu zaplanowano oraz zaprojektowano 2 scenariusze, koncentrujące się na dwóch modelach shardingu: horyzontalnym oraz wertykalnym, z wykorzystaniem replikacji i komponentów systemu Vitess, takich jak VTGate i VTablet.

Ze względu na potrzeby sprzętowe przewyższające dostępne zasoby (zarówno prywatne w postaci pamięci własnych maszyn jak i wirtualne w postaci chmury AWS) konieczna była rezygnacja z zaplanowanych scenariuszy skalowania i decyzja o zmianie scenariusza:

#### Scenariusz testu

W zaprezentowanym scenariuszu skupiliśmy się na pokazaniu działania systemu Vitess przez wywoływanie queries do bazy danych oraz obserwacji zachowania systemu z pomocą narzędzie OTel, Prometheus i Grafana.

Przebieg:

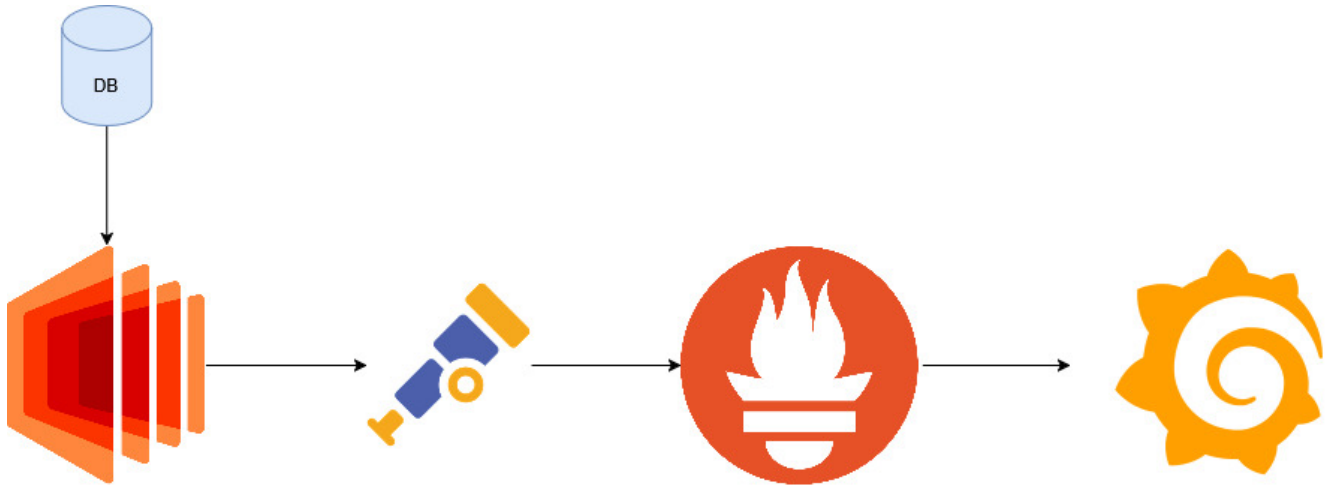
- Użytkownik posiada dostęp do bazy danych właściwie przygotowanej do działania według kroków w punktach 6-7
- Użytkownik wykonuje wielokrotnie zapytania do bazy danych obciążając tym samym system, np. `SELECT * FROM customer`, `SELECT * FROM product` czy `SELECT * FROM order` - konieczne jest przeprowadzenie zapytań w bardzo niewielkim odstępie czasowym tak, żeby odpowiednio dociążyć system
- Obserwujemy na Grafanie zwiększone wykorzystanie CPU w momencie wywołania zapytań

### 4 Architektura rozwiązania

W projekcie zaprezentowano uruchomienie klastra Vitess w środowisku Kubernetes przy użyciu operatora Vitess. Cała architektura opiera się na kilku warstwach:

- Kubernetes – jako platforma orkiestracyjna do uruchamiania i skalowania usług kontenerowych.
- Vitess Operator – komponent zarządzający zasobami Vitess w klastrze K8s (shardy, replikacje, VTGate, VTablet, itd.).
- MySQL – backend bazodanowy obsługiwany przez Vitess.
- OTEL Collector - modułowy komponent zbierający, przetwarzający i eksportujący dane telemetryczne (logi, metryki, ślady) zgodnie ze standardem OpenTelemetry
- Prometheus - system monitorowania i alertowania, który zbiera metryki z aplikacji i usług za pomocą modelu pull i przechowuje je w swojej bazie czasowej.
- Grafana – system do wizualizacji metryk.

Architektura została rozbudowana o integrację z narzędziami do obserwowalności (observability), dzięki czemu możliwe było przeanalizowanie rozkładu zapytań i obciążeń w czasie rzeczywistym.



## 5 Konfiguracja środowiska

Środowisko wykonawcze projektu zostało uruchomione lokalnie przy użyciu Minikube — lekkiej wersji klastra Kubernetes przeznaczonej do uruchamiania na pojedynczej maszynie. W celu zapewnienia odpowiednich zasobów dla komponentów Vitessa, klaster został zainicjowany za pomocą następującej komendy:

```
minikube start --cpus=8 --memory=4096 --disk-size=50g
```

Konfiguracja ta przydziela klastrowi 8 wirtualnych CPU, 4 GB pamięci RAM oraz 50 GB przestrzeni dyskowej.

## 6 Metody instalacji

Aby móc korzystać z wyżej opisanego demo, należy wcześniej zainstalować następujące narzędzia:

- kubectl v1.30.2 – narzędzie wiersza poleceń służące do zarządzania klastrami Kubernetes. Umożliwia wykonywanie operacji takich jak wdrażanie aplikacji, monitorowanie zasobów oraz diagnozowanie problemów w środowisku Kubernetes.

- `mysql v5.7` – narzędzie wiersza poleceń umożliwiające łączenie się z serwerem MySQL, wykonywanie zapytań SQL oraz zarządzanie bazami danych. Jest przydatne do testowania połączeń, przeglądania danych i administracji bazą.
- `vtctldclient` – narzędzie wiersza poleceń służące do komunikacji z komponentem `vtctld` w systemie Vitess. Umożliwia zarządzanie shardami, `keyspace`’ami i innymi elementami klastra Vitess.
- `Docker` – platforma do tworzenia, uruchamiania i zarządzania kontenerami. W projekcie wykorzystywana jest do budowania obrazów oraz uruchamiania komponentów systemu Vitess, Otel oraz Grafana w środowisku kontenerowym.
- `Minikube` – narzędzie umożliwiające lokalne uruchomienie klastra Kubernetes. Używane w tym projekcie jako środowisko testowe dla Vitessa.

## 7. Jak odtworzyć projekt krok po kroku

Sekcja ta ma za zadanie umożliwić innej osobie dokładne odtworzenie środowiska od zera, w tym instalacji narzędzi i ich konfiguracji. Przedstawia pełny „przepis” krok po kroku.

### Setup minikube

```
minikube start --cpus=8 --memory=4096 --disk-size=50g
```

### Setup namespace

```
kubectl create namespace example
kubectl create namespace telemetry
```

### Setup Vitess

```
cd kube
kubectl apply -f operator.yaml # poczekać aż wstanie
kubectl apply -f 101_initial_cluster.yaml # poczekać aż wstanie
```

### Seed bazy danych + Port Forwarding

```
./pf.sh &
alias mysql="mysql -h 127.0.0.1 -P 15306 -u user"
alias vtctldclient="vtctldclient --server localhost:15999 --alsologtostderr"
vtctldclient ApplySchema --sql="$(cat create_commerce_schema.sql)" commerce
vtctldclient ApplyVSchema --vschema="$(cat vschema_commerce_initial.json)"
commerce
```

## Setup narzędzi do Otel'a

```
cd otel
find . -name '*.yaml' -exec kubectl apply -f {} \; # zaaplikowanie kazdego pliku
yaml
kubectl port-forward svc/grafana 3000:80 -n telemetry # port forwarding grafany
```

## Dodanie Prometheusa jako źródło danych

1. Przejdź do zakładki Connections.
2. Wybierz Data Source.
3. Kliknij Add new data source.
4. Z listy dostępnych źródeł wybierz Prometheus.
5. W polu URL wpisz adres: `http://prometheus.telemetry.svc.cluster.local:9090`
6. Kliknij przycisk Save & test, aby zapisać ustawienia i przetestować połączenie.

## Przetestowanie dema

```
# trzeba byc w ./kube
for i in {1..10000}; do mysql --table < select_commerce_data.sql > /dev/null; done
```

## 8. Otrzymane wyniki

W Grafanie testowaliśmy metrykę o nazwie `container_cpu_usage_seconds_total`, która służy do mierzenia zużycia CPU przez kontenery. Metryka ta pozwala na dokładną analizę obciążenia CPU generowanego przez poszczególne komponenty systemu uruchomione w kontenerach.

W naszym eksperymencie zebraliśmy dane dla dwóch różnych scenariuszy:

- Bez obciążenia bazy danych – w tym trybie system działał w stanie spoczynku, bez generowania dodatkowego ruchu ani zapytań do bazy danych. Pomiar zużycia CPU w tym scenariuszu pozwala ocenić podstawowe zużycie zasobów przez Vitess oraz powiązane komponenty, co stanowi punkt odniesienia do dalszych analiz.
- Z obciążeniem bazy danych – w tym scenariuszu generowaliśmy ruch i zapytania do bazy danych, aby zasymulować rzeczywiste użytkowanie systemu. Dzięki temu mogliśmy zaobserwować, jak wzrasta zużycie CPU w odpowiedzi na obciążenie oraz ocenić wydajność i skalowalność systemu Vitess pod większym natężeniem pracy.

Poniżej prezentujemy wykresy i dane zebrane z monitoringu, które ilustrują różnice w zużyciu CPU między tymi dwoma stanami, co pozwala na lepsze zrozumienie wpływu obciążenia na zasoby systemowe.

Data sourceprometheus

Query optionsMD = auto = 1156Interval = 15s

Query inspector

A(prometheus)

Kick start your query

Explain

Run queries

Builder

Code

Metric

Label filters

container\_cpu\_usage\_seconds\_total

namespace = example

1

```
rate(container_cpu_usage_seconds_total{namespace="example"}[1m]) * 100
```

Fetch all series matching metric name and label filters.

Rate

Multiply by scalar

+ Operations

Range

Value

2

```
rate(<expr>[1m])
```

Calculates the per-second average rate of increase of the time series in the range vector. Breaks in monotonicity (such as counter resets due to target restarts) are automatically adjusted for. Also, the calculation extrapolates to the ends of the time range, allowing for missed scrapes or imperfect alignment of scrape cycles with the range's time period.

3

```
<expr> * 100
```

no docs

```
rate(container_cpu_usage_seconds_total{namespace="example"}[1m]) * 100
```

Options

Legend

Min step

Format

Type

Exemplars

{{pod}}

auto

Time series

Range

Instant

Both

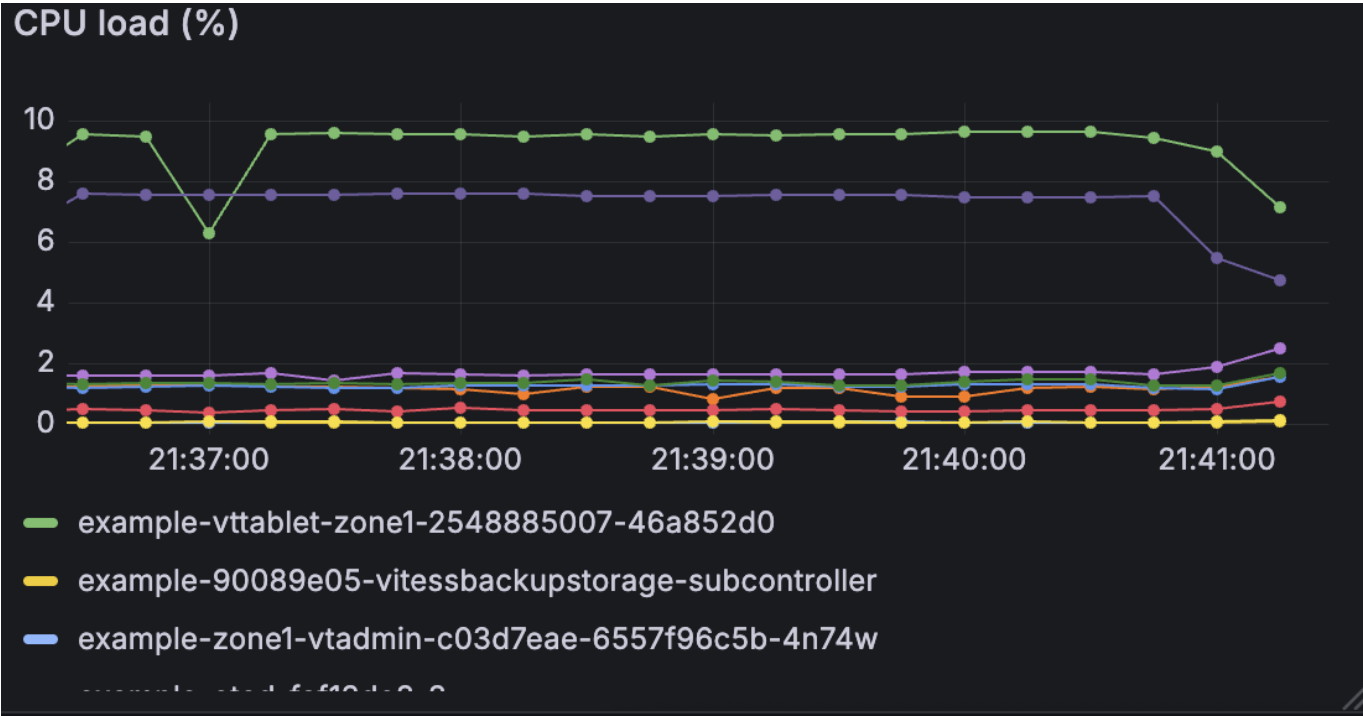
### CPU load (%)

Time	example-vttablet-zone1-2548885007-46a852d0	example-90089e05-vitessbackupstorage-subcontroller	example-zone1-vtadmin-c03d7eae-6557f96c5b-4n74w
21:31:00	3.5	0.1	1.9
21:31:30	3.4	0.1	1.8
21:32:00	3.5	0.1	1.8
21:32:30	3.5	0.1	1.8
21:33:00	3.5	0.1	1.8
21:33:30	3.5	0.1	1.8
21:34:00	3.6	0.1	1.9

Legend:

- example-vttablet-zone1-2548885007-46a852d0
- example-90089e05-vitessbackupstorage-subcontroller
- example-zone1-vtadmin-c03d7eae-6557f96c5b-4n74w

6 / 12



9. Wykorzystanie narzędzi AI

W trakcie tworzenia projektu zasięgałiśmy pomocy narzędzi takich jak ChatGPT czy DeepSeek zarówno w celu znalezienia poprawnych komend tworzących kolejne komponenty systemu jak i dla celów przyspieszenia pracy przez generowanie mock'ów/prostych komend.

Modele sztucznej inteligencji dodają dodatkowo bardzo dokładne wytłumaczenie kolejnych kroków, na przykład w tych przypadku:



dlatego zrzuty ekranu obrazujące wykorzystanie modeli są docięte do pytań/odpowiedzi.

Przedstawione narzędzia często miały rację, szczególnie w przypadku najczęściej używanych rozwiązań jak kubernetes czy mysql - często konieczne było doprecyzowanie dodatkowych parametrów, lecz komendy zwracane przez program były prawidłowe:



jak uruchomić lokalnie środowisko Minikube?

mam maca i chce wykorzystać ok. 4gb pamięci

**Przykład z podaniem drivera Docker (zalecane na macOS):**

bash

Copy Edit



```
minikube start --driver=docker --cpus=8 --memory=4096 --disk-size=50g
```

Również w przypadku samego Vitessa, chatGPT zdarzał się być przydatny:

## 4. Zastosuj manifesty

W katalogu z manifestami masz wiele plików `.yaml`. Najprościej:

bash



 Copy  Edit

```
kubectl apply -f .
```

To wdroży wszystkie zasoby — StatefulSets, ConfigMaps, Services itd.

## 5. Sprawdź status podów

bash

 Copy  Edit



```
kubectl get pods -n vitess
```

Poczekaj, aż wszystkie będą w stanie `Running`.

## 6. Port-forward do VTGate

VTGate to warstwa dostępu do bazy Vitess. Aby się połączyć:

bash

 Copy  Edit

```
kubectl port-forward -n vitess svc/vtgate 15306:15306
```

, lecz jego przydatność zaczynała się dopiero w momencie kiedy zadało się właściwe pytanie, np. "Jak postawić Vitessa z yamla". Proste pytanie "Jak uruchomić Vitessa" prowadziło do błędnych informacji:

## 1. Przygotuj środowisko

- Minikube z co najmniej 4 CPU i 8 GB RAM (lub więcej, zależnie od potrzeb)
- `kubectl` skonfigurowane i połączone z Twoim klastrem
- `helm` — menedżer pakietów Kubernetes

Instalacja Helm:

```
bash
```

[Copy](#) [Edit](#)

```
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
```

## 2. Pobierz repozytorium Vitess Helm Chart

Vitess oficjalnie wspiera instalację przez Helm.

```
bash
```

[Copy](#) [Edit](#)

```
git clone https://github.com/vitessio/vitess.git  
cd vitess
```

, ponieważ Vitess nie jest już wspierany przez helm. Oczywiście w momencie zwrócenia na to uwagi, informacja staje się dla niego oczywista:

vitess już nie jest wspierany przez helm

Masz całkowitą rację — Vitess Helm Chart został wycofany i nie jest już oficjalnie wspierany. Od 2023 r. zespół Vitess zaleca instalację tylko przez operatora Vitess (`vitess-operator`) lub ręcznie przez YAML-e.

Tak więc podsumowując wykorzystanie narzędzi AI, są to bardzo wygodne rozwiązania, jednak dopiero wtedy, kiedy jest się obeznanym w temacie i intuicyjnie wie się, kiedy odpowiedzi mają szansę być poprawne, a kiedy nie. Dopóki nie jest się dobrze zaznajomionym z technologią, najlepszym wyborem jest skorzystanie z docsów.

## 10. Wnioski

W przeprowadzonym eksperymencie, którego celem było uruchomienie środowiska Vitess oraz monitorowanie jego działania za pomocą OpenTelemetry, Prometheusa i Grafany, udało się uzyskać szczegółowe informacje na temat zużycia zasobów przez poszczególne komponenty klastra. Analiza metryk pokazała, że największe obciążenie procesora podczas wykonywania zapytań SQL przypada na komponenty vtablet oraz vtgate. vtablet odpowiada za bezpośredni kontakt z bazą danych MySQL, a vtgate pełni funkcję bramy agregującej zapytania i rozdzielającej je do odpowiednich shardów – ich intensywna praca w czasie przetwarzania zapytań jest więc naturalna, ale też stanowi kluczowy punkt obserwacyjny pod kątem optymalizacji.

Z eksperymentu wynikają również inne istotne wnioski – Vitess okazuje się być bardzo zasobożerny, szczególnie jeśli chodzi o zapotrzebowanie na pamięć RAM. Ta obserwacja nie wynika bezpośrednio z wykresów metryk, lecz z praktycznych problemów napotkanych podczas prób uruchomienia klastra Vitess na sprzęcie o ograniczonych zasobach. Próby postawienia środowiska na maszynach z niewielką ilością dostępnej pamięci często kończyły się niepowodzeniem lub niestabilnym działaniem usług. Co więcej, zauważalny jest niedobór przykładów użycia Vitessa w internecie, co utrudnia jego wdrażanie i rozwiązywanie problemów.

Prometheus i Grafana wyróżniają się bardzo dobrą dokumentacją oraz szeroką kompatybilnością z różnymi systemami i usługami. Ich elastyczność oraz bogata kolekcja gotowych dashboardów sprawiają, że integracja z Vitess – choć wymaga pewnej konfiguracji – przebiega sprawnie i pozwala na szybkie uzyskanie wartościowych danych

## 11. Bibliografia

- Vitess: <https://vitess.io/docs>
- OpenTelemetry: <https://opentelemetry.io/docs>
- Grafana: <https://grafana.com/>
- Kubernetes: <https://kubernetes.io/>
- Mysql: <https://www.mysql.com/>
- Minikube: <https://minikube.sigs.k8s.io/docs/start/>
- Docker: <https://www.docker.com/>