КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

# Дослідження стратегій кешування у високонавантажених .NET-додатках


ВИКОНАЛА:

**Студентка 4-го курсу**

**Дяченко В.Ю.**

КЕРІВНИК:

**Борозенний С.О.**

# ЧОМУ СТАТИЧНИЙ TTL — ЦЕ ПРОБЛЕМА?

### Жорсткий компроміс

Неможливість знайти баланс між актуальністю даних та навантаженням на БД без ручного втручання.

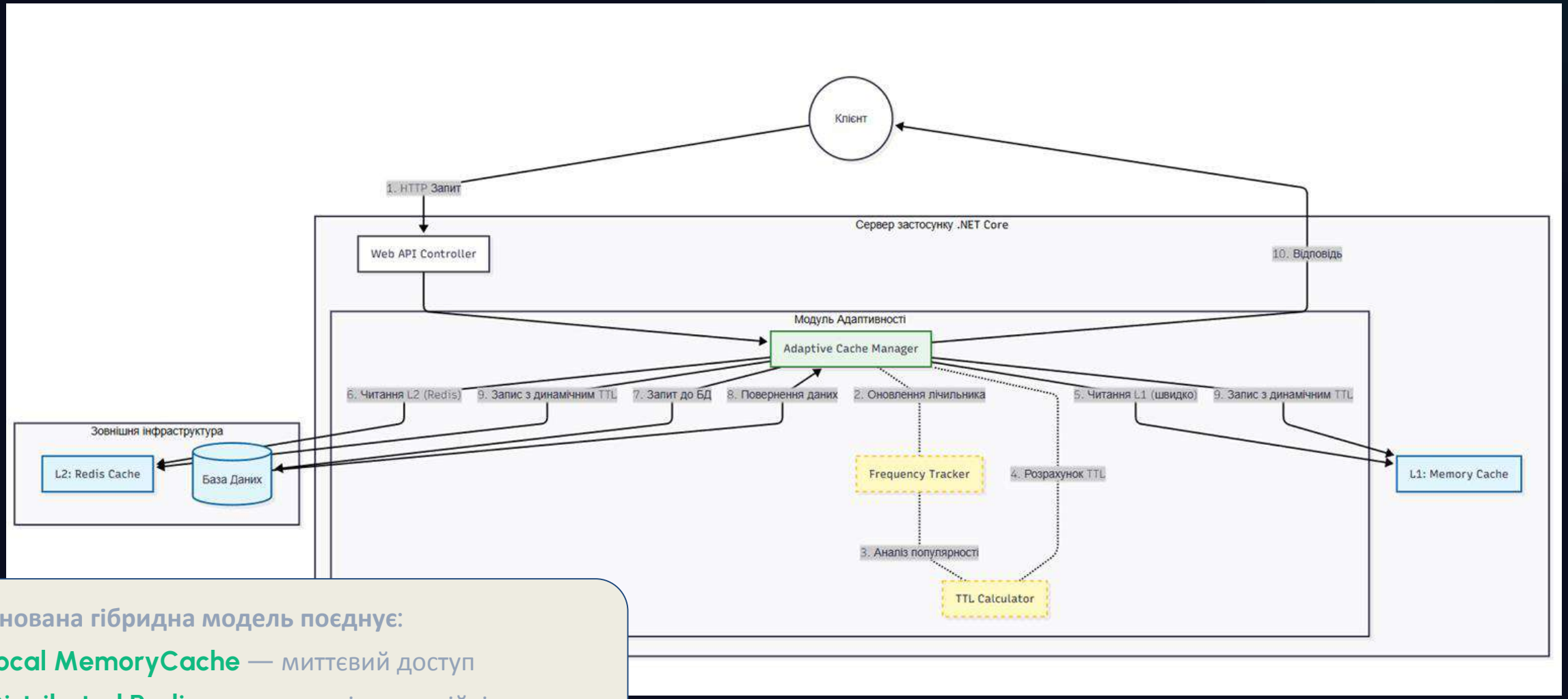
### «Холодні» дані

Непопулярні об'єкти безпідставно займають оперативну пам'ять до повного закінчення TTL.

### Cache Stampede

Ризик лавиноподібного навантаження на джерело даних при одночасному оновленні популярних ключів.

# АРХІТЕКТУРА АДАПТИВНОЇ СИСТЕМИ




Запропонована гібридна модель поєднує:

- **L1: Local MemoryCache** — миттєвий доступ
- **L2: Distributed Redis** — узгодженість та стійкість
- **Adaptive Core** — інтелектуальний аналіз популярності

# МАТЕМАТИЧНА МОДЕЛЬ SLIDING WINDOW

Умова фільтрації запитів:

$$T_{now} - T_{e} > \Delta t$$

Розрахунок динамічного TTL:

$$TTL_{raw} = TTL_{base} + (S(k) \cdot K)$$

**Механізм Decay:** Проактивне скорочення часу життя для даних, популярність яких стрімко падає.

**Механізм Jitter:** Захист від одночасної інвалідації кешу через випадковий зсув часу видалення (±5%)

# ПРОГРАМНА РЕАЛІЗАЦІЯ

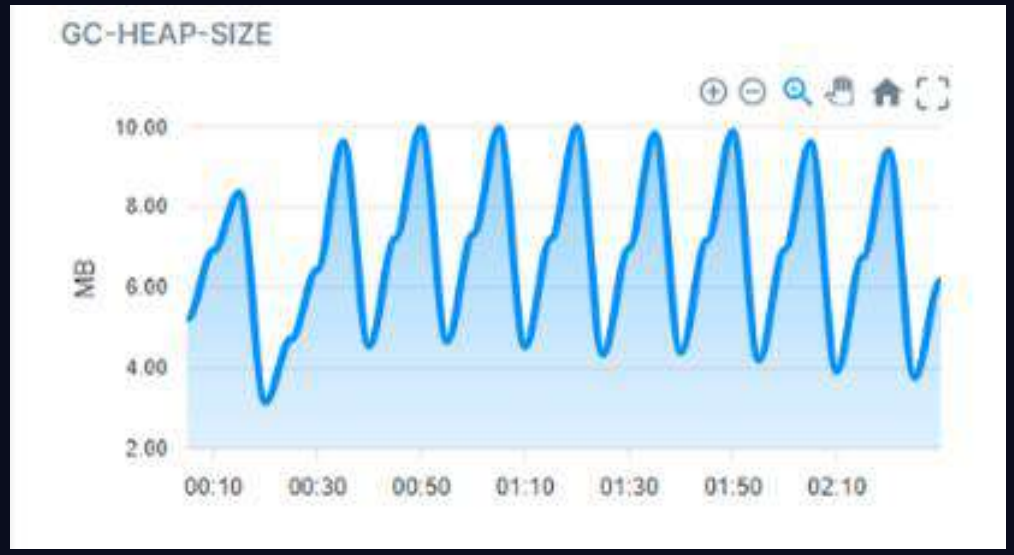




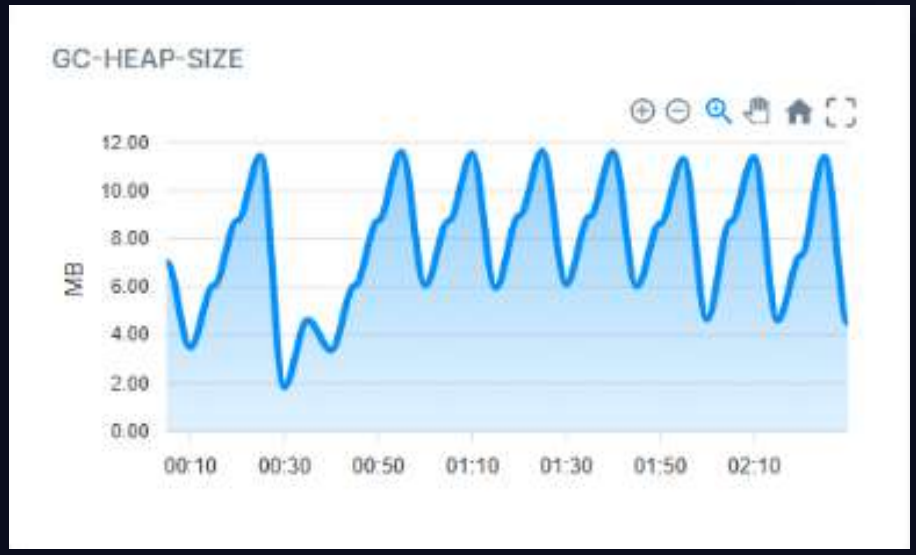

```csharp
public TimeSpan CalculateTtl(double currentScore, TimeSpan? previousTtl = null)
{
    double finalSeconds;

    if (currentScore < ThresholdLow && previousTtl.HasValue)
    {
        finalSeconds = previousTtl.Value.TotalSeconds * DecayFactor;
    }
    else
    {
        finalSeconds = _minTtl.TotalSeconds + (currentScore * K);
    }

    var jitter = 1 + (Random.Shared.NextDouble() * 2 - 1) * JitterRange;
    finalSeconds *= jitter;

    finalSeconds = Math.Clamp(finalSeconds, _minTtl.TotalSeconds, _maxTtl.TotalSeconds);

    return TimeSpan.FromSeconds(finalSeconds);
}
```

# ВПЛИВ НА GARBAGE COLLECTOR (GC)

Статичний підхід (Cache Thrashing)




Адаптивний алгоритм (Decay)




**Конфігурація тестового стенда: AMD Ryzen 7 5800H, 16 GB RAM, .NET 8.0 та Redis 8.0 у Docker-контейнері**

# РЕЗУЛЬТАТИ ТЕСТУВАННЯ (PARETO)

Статичний підхід:

```
requests: all = 6300, ok = 6300
latency (ms): mean = 307.66, max = 859.38
latency percentile (ms): p50 = 503.04
```

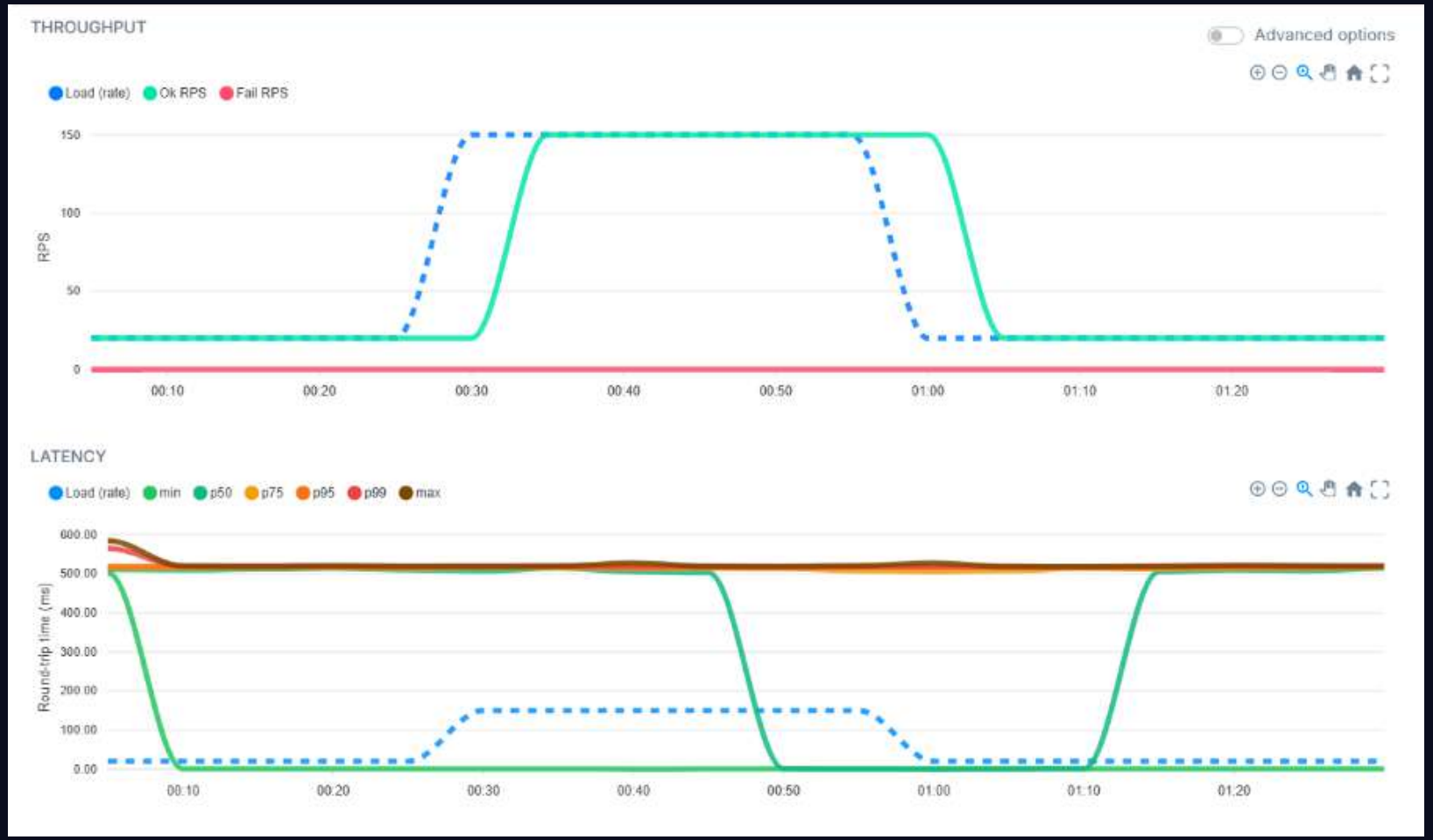
Адаптивний алгоритм:

```
requests: all = 7500, ok = 7500, RPS = 50/s
latency (ms): mean = 139.99, max = 518.86
latency percentile (ms): p50 = 0.52
```

| Метрика (мс) | Static | Adaptive |
|---|---|---|
| Min Latency | 0.19 | 0.26 |
| Mean Latency | 307.66 | **139.99** |
| Max Latency | 859.38 | **518.86** |
| p50 (Медіана) | 503.04 | 0.52 |

**Конфігурація тестового стенда: AMD Ryzen 7 5800H, 16 GB RAM, .NET 8.0 та Redis 8.0 у Docker-контейнері**

# СТРЕС-ТЕСТ «ЧОРНА П'ЯТНИЦЯ»




**0 помилок (Fail = 0)** при стрибку навантаження до 150 RPS.

**Mean Latency = 306.15 мс** (стабільна робота під піком).

**p50 = 504.32 мс** — відсутність лавиноподібного зростання черг.

# КІЛЬКІСНІ РЕЗУЛЬТАТИ

### -40%

Скорочення максимальної затримки (з 859 мс до **518.86 мс**).

### 0.52 мс

Медіанна затримка. Понад 50% трафіку обслуговується миттєво з L1 кешу.

### Стабільність

Середня затримка при піку 150 RPS склала **306.15 мс** при нульовій кількості помилок.

# Дякую за увагу!

Кваліфікаційна робота на тему: «Дослідження стратегій кешування у високонавантажених .NET-додатках»
Виконала: Дяченко Владислава