언리얼 페스트 2024 서울

드래곤 소드

데디케이티드 서버 최적화와 비용절감 사례

김형락
엔진프로그래머
HOUND13

발표자 소개

NCSoft

프로젝트 혼(GSTAR 2014)
리드 프로그래머

WonderPeople

슈퍼피플
리드 엔지니어

HOUND13

드래곤 소드
엔진 프로그래머

목차

Dragon Sword 소개

AWS 비용 분석 및 최적화 전략

EC2 인스턴스 요금 옵션

온디맨드 인스턴스

중단 없는 인스턴스
스팟 인스턴스 대비 높은 비용

스팟 인스턴스

온디맨드 인스턴스 대비 대폭 낮은 요금
5~20% 수준의 인스턴스 중단

인스턴스 중단에 대한 대응 필요

EC2 인스턴스 운영 체제

WINDOWS

익숙한 개발 환경
리눅스 대비 높은 비용

LINUX

Windows 대비 50% 낮은 비용
Third party library Integration

WSL등 별도의 개발 환경 구성 필요.

EC2 인스턴스 유형

필요 스펙에 따라 적절한 인스턴스 타입 결정

2Core 4GiB
(c??.large)

2Core 8GiB
(m??.large)

4Core 8GiB
(c??.xlarge)

4Core 16GiB
(m??.xlarge)

8Core 16GiB
(c??.2xlarge)

EC2 Auto Scaling

- 접속 유저 수에 따른
  동적 인스턴스 관리
- GameLift FleetIQ
- 온디맨드 or 스팟 선택 가능

EC2 Traffic

- 수신 비용 무료
- 송신 비용 사용량 만큼 증가

AWS 비용 분석 정리

WINDOWS → LINUX

Auto Scaling 사용

온디맨드 → 스팟 인스턴스

최적화 우선 순위

1. 성능 최적화
서버당 유저 수 증가

2. 메모리 최적화
메모리 사용량을 2^N이하로 줄이기

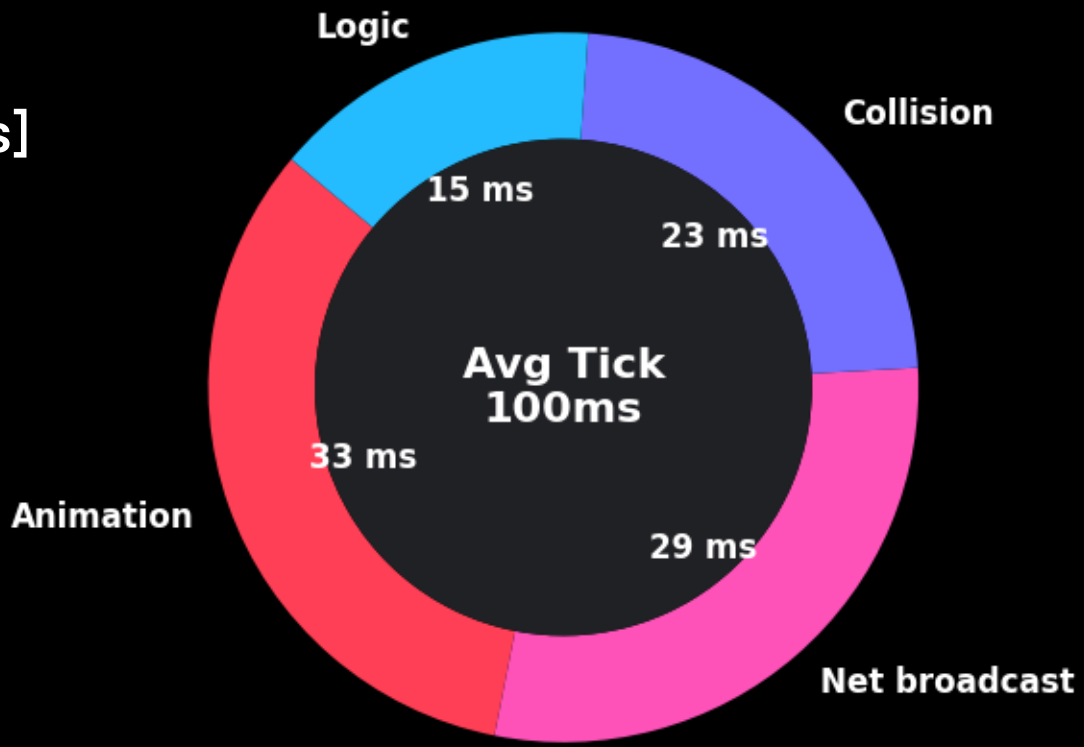
3. 네트워크 최적화
송신 데이터 줄이기

성능 최적화

데디케이티드 서버
CPU 성능 프로파일링(초기)

AWS c5.2xlarge[windows]
4 플레이어 40 몬스터

Avg Tick 100ms
- Animation 33ms
- Net broadcast 29ms
- Collision 23ms
- Logic 15ms



Animation 최적화

- AnimNotify 의존적 액션 시스템
- AnimNotify는 애니메이션 Tick에 의존
- 기존 리소스와 구조 변경없이 최적화 할 수 있을까?

Action Instance

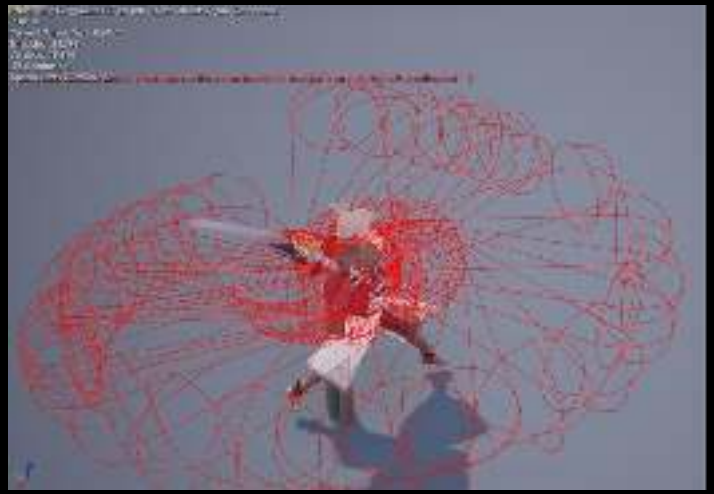
```
void FDsActionInstance::ProcessAnimNotifies(float DeltaSeconds, UAnimSequenceBase* AnimSequenceBase)
{
    for (const FAnimNotifyEvent& Event : AnimSequenceBase->Notifies)
    {
        UDsBaseAnimNotify* AnimNotify = Cast<UDsBaseAnimNotify>(Event.Notify);
        if (IsValid(AnimNotify))
        {
            if (LastElapsedTime <= Event.GetTime() && Event.GetTime() < ElapsedTime)
            {
                AnimNotify->AnimNotify(DeltaSeconds, ActionComponent->GetMeshComponent(), AnimSequenceBase);
            }
        }
    }
```

✓ 기존 AnimNotify 시스템은 입력 데이터로 사용

✓ Action Instance Tick에서 AnimNotify를 직접 호출

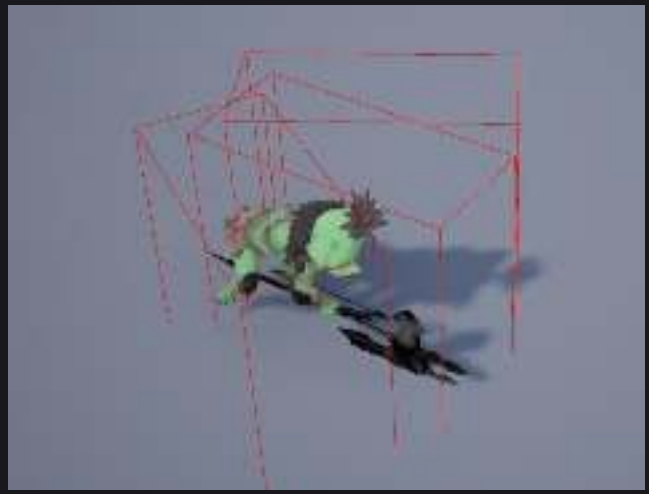
✓ 데디케이티드 서버는 Animation Tick 끔

ActionInstance Off



- Bone 의존적 sweep test

ActionInstance On



- Bone 독립적 Overlap Test

Collision 최적화

OverlapTest대신 World2DGrid를 활용

```
//TArray<FOverlapResult> OverlapResults;
//GetWorld()->OverlapMultiByProfile(OverlapResults, OwnerLocation, FQuat::Identity, DCollisionProfileName_OffenceBound, OverlapShape);

const UDGameInstance* GameInstance = Cast<UDGameInstance>( Src: UGameplayStatics::GetGameInstance( WorldContextObject: this));
const FDsWorldGrid2D* WorldGrid2D = IsValid(GameInstance)? GameInstance->GetWorldGrid2D() : nullptr;
TSet<TObjectPtr<AActor>> CombatTargets;
WorldGrid2D->GatherActorsInRange(OwnerLocation, InRadius: OwnerFieldCharacter->GetMaxSenseRange(), [&] CombatTargets);
```

Net broadcast 최적화

✓ 액터 순환 비용

- 클라이언트 연결마다 액터를 순환하며 Relevant 검사.

✓ Recent Buffer 조회

✓ Net serialization

ReplicationGraph

네트워크 액터를 Cell 단위의 그리드로 구조화, 정책에 따라
Static/Dynamic/Dormant/Dependent 등으로 구분하여 노드로 관리

전체 ReplicationActor 순회(X)

↓

클라이언트 연결마다 Cell 단위로 Relevant 검사

클라이언트 Viewer의 방향/거리에 따라
상대적 NetFrequency 적용

ReplicationGraph

액터의 특징에 따라 정책 설정

```
void UDsReplicationGraph::SetupGameClassSettings()
{
    // not routed
    SetRepNodePolicy(ADBaseActor::StaticClass(), EClassRepNodeMapping::NotRouted);
    SetRepNodePolicy(ADecalActor::StaticClass(), EClassRepNodeMapping::NotRouted);
    SetRepNodePolicy(ADBaseVolume::StaticClass(), EClassRepNodeMapping::NotRouted);

    // dynamic
    SetRepNodePolicy(ADsBaseProjectile::StaticClass(), EClassRepNodeMapping::Spatialize_Dynamic);

    // static
    SetRepNodePolicy(ADsAnimationProp::StaticClass(), EClassRepNodeMapping::Spatialize_Static);
    SetRepNodePolicy(ADETSwitchBaseActor::StaticClass(), EClassRepNodeMapping::Spatialize_Static);

    SetRepNodePolicy(ADsWeaponActor::StaticClass(), EClassRepNodeMapping::NotRouted); // depend on character
```

ReplicationGraph

액터 종속관계 설정을 통한 순환 비용 감소

무기 클래스를 캐릭터 클래스에 종속관계로 설정

```
void ADsFieldCharacter::OnEquipWeapon(ADsWeaponActor* WeaponActor)
{
    FNetDelegates::OnAddNetDependentActor.Broadcast(this, WeaponActor);
}

void ADsFieldCharacter::OnUnEquipWeapon(ADsWeaponActor* WeaponActor)
{
    FNetDelegates::OnRemoveNetDependentActor.Broadcast(this, WeaponActor);
}
```

ReplicationGraph

클라이언트 Viewer의 방향/거리에 따라
상대적 NetFrequency 적용

```
void UDsReplicationGraph::SetRepPeriod()
{
    UReplicationGraphNode_DynamicSpatialFrequency::DefaultSettings.ZoneSettings_NonFastSharedActors[Rear].MinRepPeriod = 7;
    UReplicationGraphNode_DynamicSpatialFrequency::DefaultSettings.ZoneSettings_NonFastSharedActors[Rear].MaxRepPeriod = 8;
    UReplicationGraphNode_DynamicSpatialFrequency::DefaultSettings.ZoneSettings_NonFastSharedActors[Side].MinRepPeriod = 5;
    UReplicationGraphNode_DynamicSpatialFrequency::DefaultSettings.ZoneSettings_NonFastSharedActors[Side].MaxRepPeriod = 8;
    UReplicationGraphNode_DynamicSpatialFrequency::DefaultSettings.ZoneSettings_NonFastSharedActors[Front].MinRepPeriod = 3;
    UReplicationGraphNode_DynamicSpatialFrequency::DefaultSettings.ZoneSettings_NonFastSharedActors[Front].MaxRepPeriod = 8;
}
```

Tick 최적화

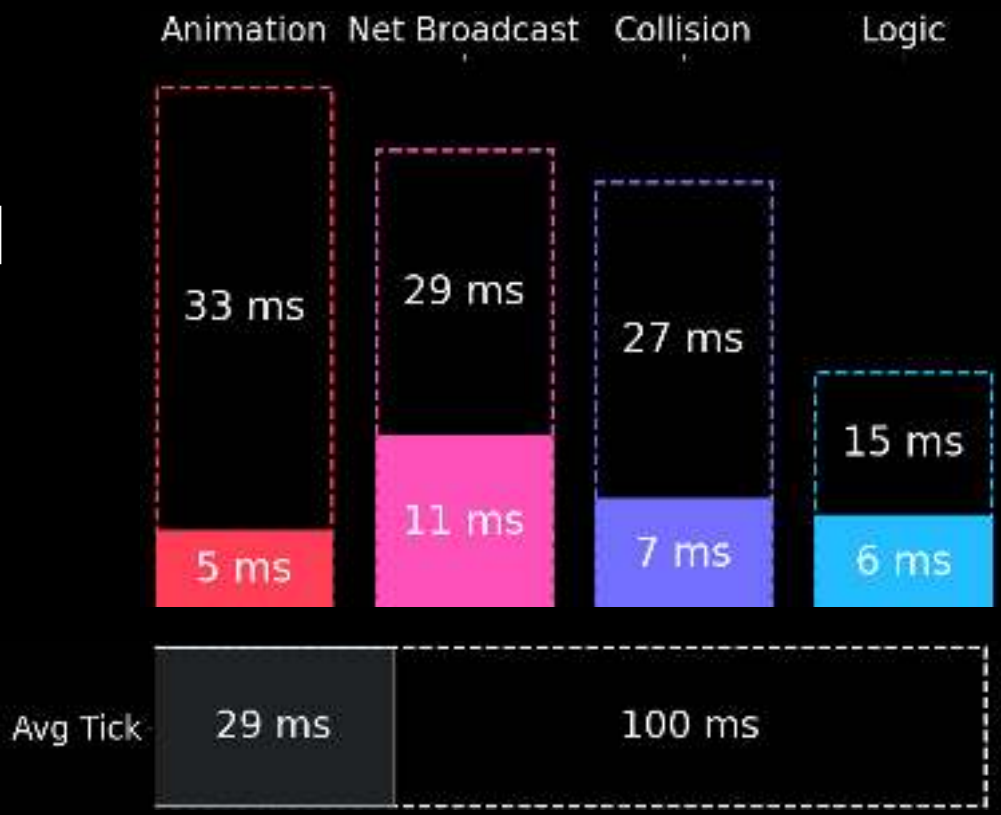
- FActorTickFunction
- FActorComponentTickFunction

bCanEverTick = false
bAllowTickOnDedicatedServer = false
bStartWithTickEnabled = false
SetTickFunctionEnable(false)
TickInterval = 지정

데디케이티드 서버
CPU 성능 프로파일링(후기)

AWS c5.2xlarge[windows]
8 플레이어 100 몬스터 (2x↑)

Avg Tick 29ms
- Animation 5ms
- Net broadcast 11ms
- Collision 7ms
- Logic 6ms



네트워크 최적화

Client 이동패킷 전송 주기 제어

GameNetworkManager.h 변수 조정

```
UPROPERTY(GlobalConfig)
float ClientNetSendMoveDeltaTime;

/** ClientNetSendMoveDeltaTimeThrottled is used in
UPROPERTY(GlobalConfig)
float ClientNetSendMoveDeltaTimeThrottled;

/** ClientNetSendMoveDeltaTimeStationary is used w
UPROPERTY(GlobalConfig)
float ClientNetSendMoveDeltaTimeStationary;
```

```
/** When player net speed (CurrentNetSpeed, based
UPROPERTY(GlobalConfig)
int32 ClientNetSendMoveThrottleAtNetSpeed;

/** When player count is greater than this amount,
UPROPERTY(GlobalConfig)
int32 ClientNetSendMoveThrottleOverPlayerCount;
```

NetSerialization

FVector_NetQuantize

2^{20} = ±1,048,576

FVector_NetQuantize10

2^{24} / 10 = ±1,677,721.6

FVector_NetQuantize100

2^{30} / 100 = ±10,737,418.24

FVector_NetQuantizeNormal

16bit -1..+1 inclusive

```
USTRUCT()
struct FDsNetVector : public FVector_NetQuantize
{
    GENERATED_USTRUCT_BODY()

    FORCEINLINE FDsNetVector()
    {}

    FORCEINLINE FDsNetVector(const FVector &InVec)
    {
        FVector::operator=(InVec);
    }
};
```

NetSerialization

언리얼 구조체에 커스텀 트레잇 적용

```
TStructOpsTypeTraitsBase2
{
    ...
    WithNetSerializer = True;
    WithNetDeltaSerializer = True
    ...
}
```

NetSerialization

구조체 Serialization 최적화 코드 예시

```
USTRUCT()
struct FDsHitEffectInfo
{
    GENERATED_BODY()

    UPROPERTY()
    FVector_NetQuantize StartPoint;

    UPROPERTY()
    FVector_NetQuantize EndPoint;

    UPROPERTY()
    FString WeaponName;

    bool NetSerialize(FArchive& Ar, class UPackageMap* Map, bool& bOutSuccess);
};
```

NetSerialization

구조체 Serialization 최적화 코드 예시

```
template<>
struct TStructOpsTypeTraits<FDsHitEffectInfo> : public TStructOpsTypeTraitsBase2<FDsHitEffectInfo>
{
    enum
    {
        WithNetSerializer = true
    };
};
```

NetSerialization

```
bool FDsHitEffectInfo::NetSerialize(FArchive& Ar, UPackageMap* Map, bool& bOutSuccess)
{
    const bool bISaving = Ar.IsSaving();
    NetSerializeOptionalValue<FVector_NetQuantize>(bISaving, [&]Ar, Value: [&]StartPoint, DefaultValue: = FVector_NetQuantize::ZeroVector, Map);
    NetSerializeOptionalValue<FVector_NetQuantize>(bISaving, [&]Ar, Value: [&]EndPoint, DefaultValue: = FVector_NetQuantize::ZeroVector, Map);
    DsCId WeaponId = 0;
    if (bISaving)
    {
        WeaponId = FDsWeaponTable::GetId(WeaponName);
    }
    Ar << WeaponId;
    if (!bISaving)
    {
        WeaponName = FDsWeaponTable::GetName(WeaponId);
    }
    return true;
}
```

디폴트 값을 자주 사용하는 경우 ➔ NetSerializeOptionalValue

FName/Fstring 등의 정적 데이터 ➔ TableId로 Serailization

Net Delta Serialization

TArray(X) → FFastArraySerializer 사용

- 규모가 큰 구조체 배열의 변수 리플리케이션에 유용

RecentBuffer 조회/비교없이

Dirty된 항목만 Outbunch에 Serialization

배열/데이터가 변경될 때 Dirty Mark 필요

+ 추가/변경 MarkItemDirty　　- 삭제 MarkArrayDirty

Net Delta Serialization

FastArraySerializer 사용 코드 예제

```
USTRUCT()
struct FDsActionCoolTime : public FFastArraySerializerItem
{
    GENERATED_BODY()

    UPROPERTY()
    DsUId ActionUId;

    UPROPERTY()
    DsCId TableId;

    UPROPERTY()
    float RemainingTime;
};
```

Net Delta Serialization

FastArraySerializer 사용 코드 예제

```
USTRUCT()
struct FDsCoolTimeContainer : public FFastArraySerializer
{
    GENERATED_BODY()

    UPROPERTY()
    TArray<FDsActorCoolTime>    Items;

    FORCEINLINE void Add(const FDsActorCoolTime& ActorCoolTime)
    {
        MarkItemDirty(Items.Add_GetRef(ActorCoolTime));
    }

    FORCEINLINE void RemoveAt(const int32 Index)
    {
        if (Index != INDEX_NONE)
        {
            Items.RemoveAt(Index, Count:1);
            MarkArrayDirty();
        }
    }
};
```

```
template<>
struct TStructOpsTypeTraits<FDsCoolTimeContainer> : public TStructOpsTypeTraitsBase2<FDsCoolTimeContainer>
{
    enum
    {
        WithNetDeltaSerializer = true,
    };
};
```

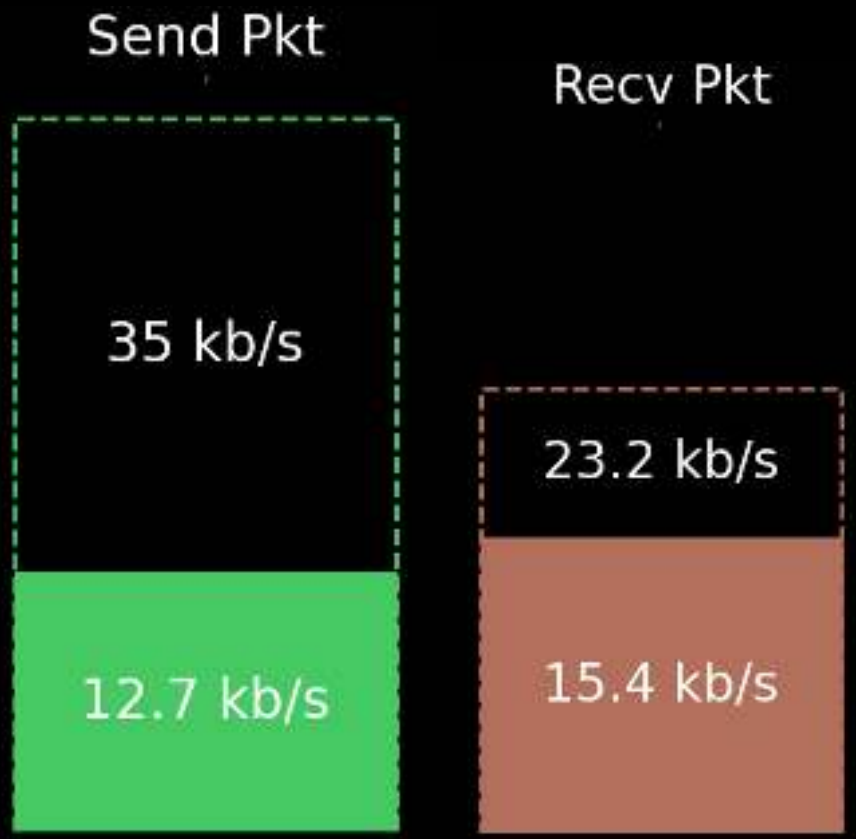
항목이 추가/삭제될 때마다
MarkDirty 관련 함수를 호출

데디케이티드 서버
네트워크 프로파일링(후기)

AWS c5.2xlarge[windows]
8 플레이어 100 몬스터 (2x↑)

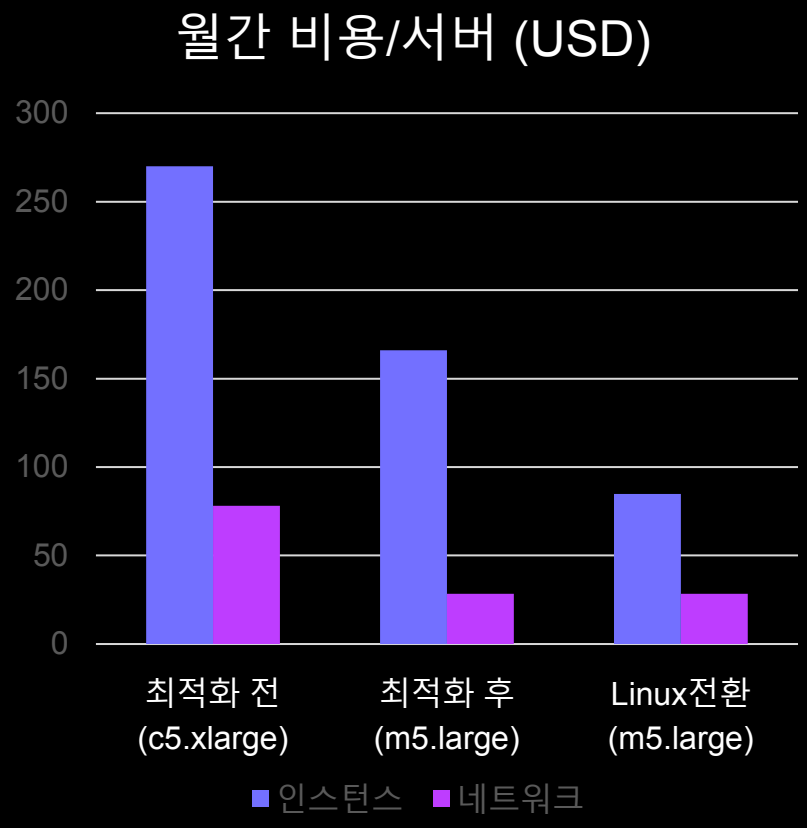
Send Pkt(peer) 35 → 12.7 kb/s
Recv Pkt(peer) 23.2 →15.4 kb/s

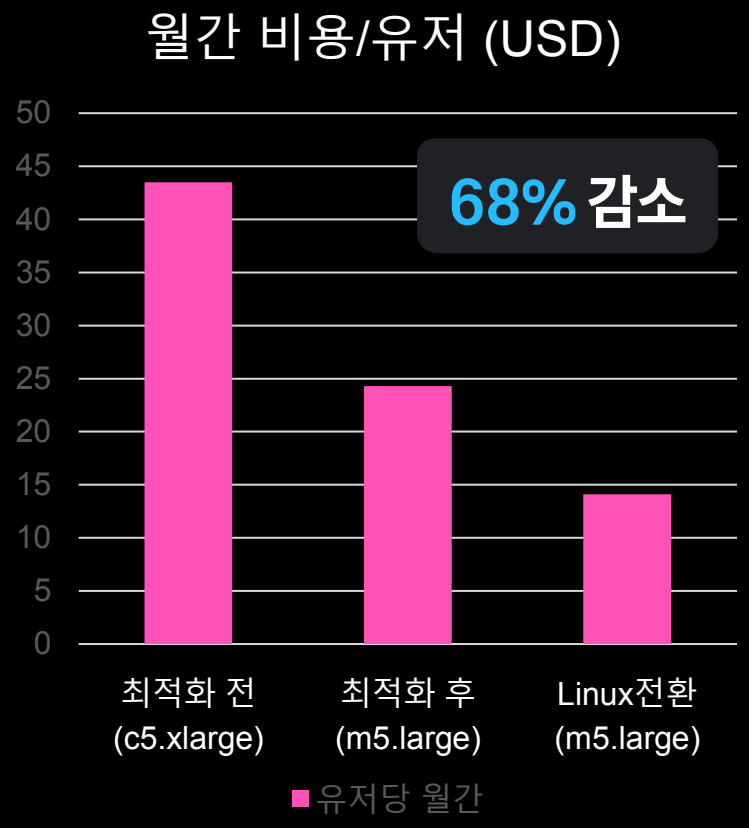
송신 트래픽 ≤60% 감소



최적화 전/후 월간 비용 변화(예상)

유저/인스턴스 : 8명





데디케이티드 서버 비용 고민

동접자수 10000명의 경우 월간 2억 비용 예상

인스턴스당 유저 수를 높이는 게 남은 과제

2개 이상 프로세스는 성능 저하 발생

단일 프로세스로 복수의 게임을 플레이?

멀티 게임 세션

멀티 게임 세션

데디케이티드 서버 프로세스 1개에 n개의 게임 세션을 운영.

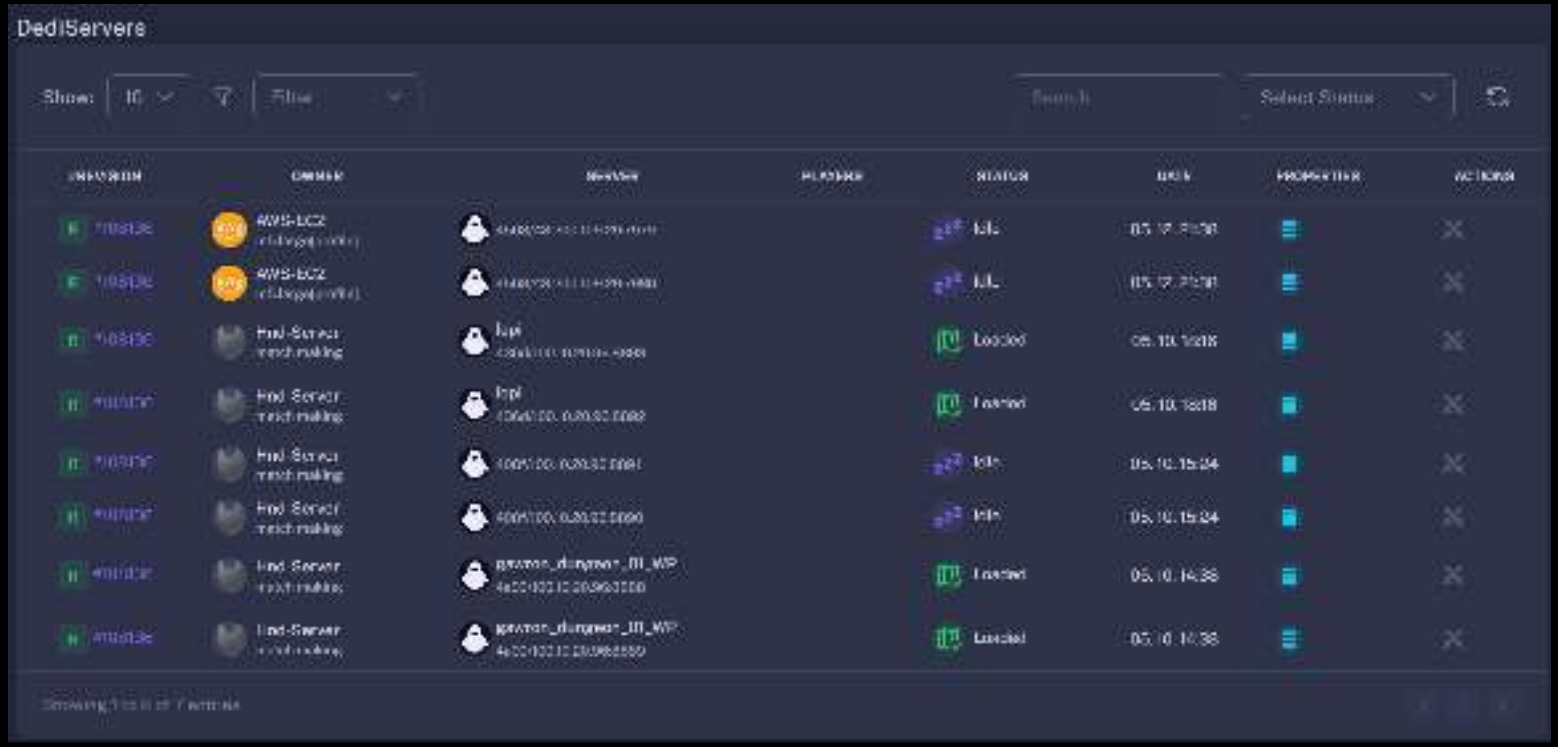
멀티 게임세션 유의 사항

게임 로직 자원이 n배로 증가.

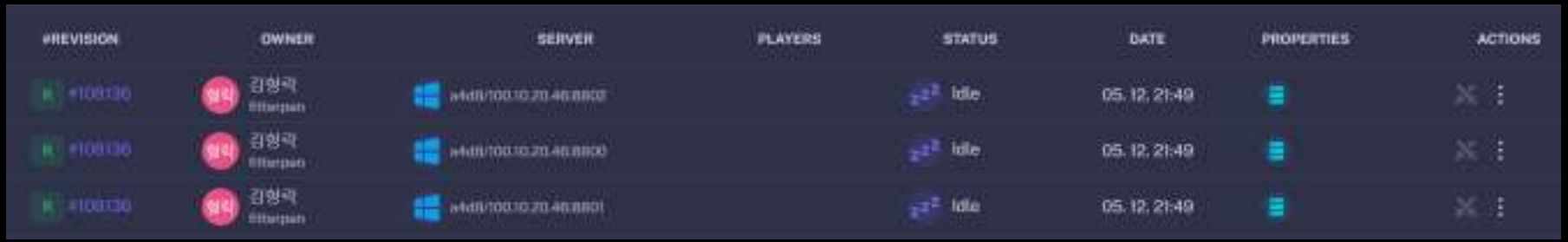
다른 게임세션에 영향을 주는 크래시/히치.

멀티 게임 세션

Devops 데디서버 관리 페이지



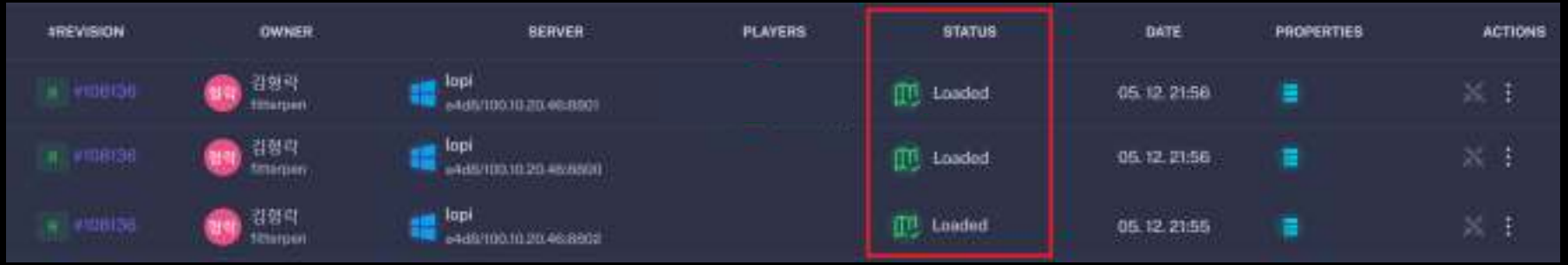
멀티 게임 세션



멀티세션 개수만큼 Entry레벨을 로딩

Idle상태 대기

멀티 게임 세션



단일 프로세스 내 모든 세션에 동일한 레벨을 로딩(메모리 최소화)

세션 수만큼 같은 리소스로 레벨을 추가 생성/로드

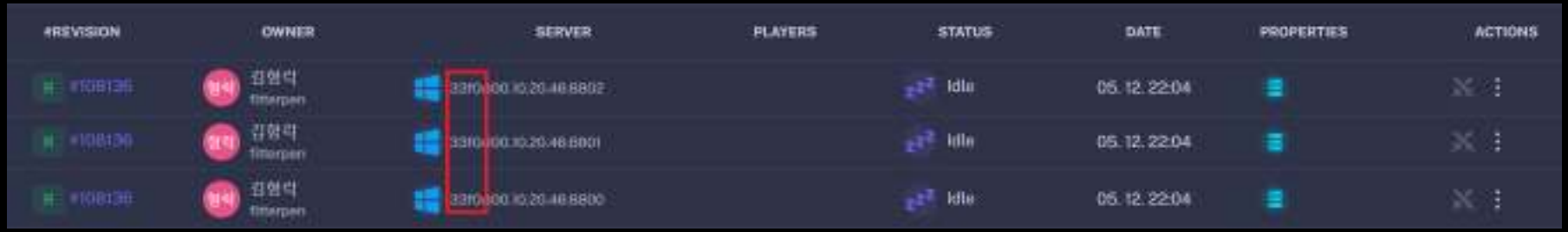
단일 프로세스 내 port는 세션마다 다르게 설정

멀티 게임 세션

매칭서버는 ip:port 정보로 분리된 게임처럼 관리

게임 종료된 세션은 폐기

멀티 게임 세션

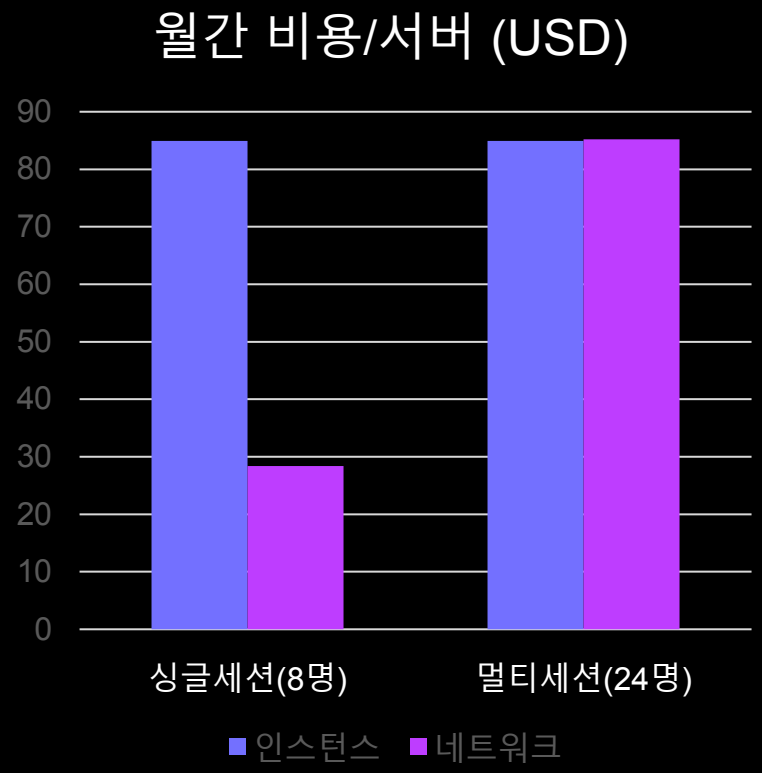


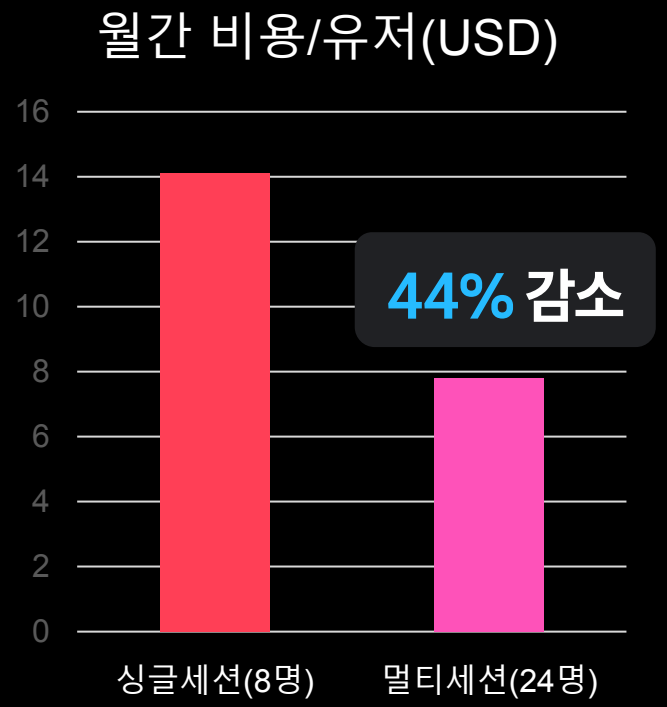
모든 세션의 게임이 종료 → 프로세스 재시작

게임세션 리셋 재사용시, 운영비용 절감 가능

멀티 세션 적용 후 월간 비용 변화 (예상)

최적화 후 m5.large(Linux)





데디케이티드 서버 비용
절감 사례

Project-S사례

UE4 기반 배틀로얄 하이퍼 슈팅 게임

8x8 km^2 오픈 월드 레벨
최대 100인 동시 플레이

스팀에서 글로벌 서비스

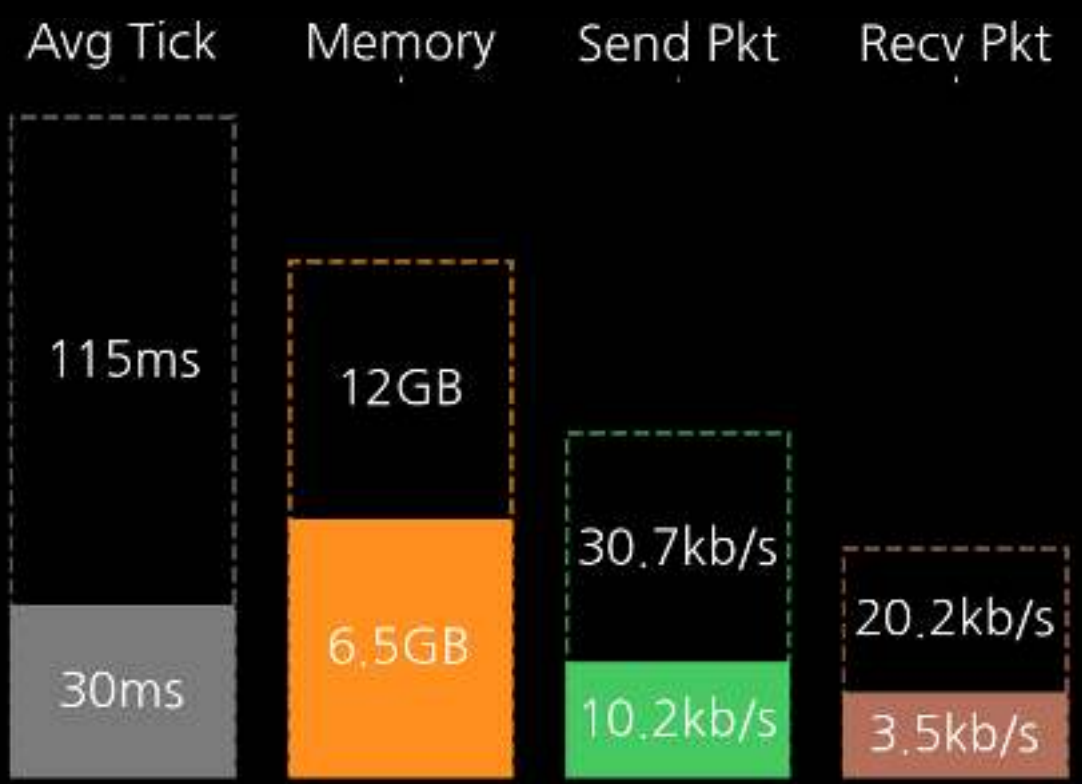
Project-S 최적화(전/후)

플레이어 10명 ➔ 100명(10x)

AWS

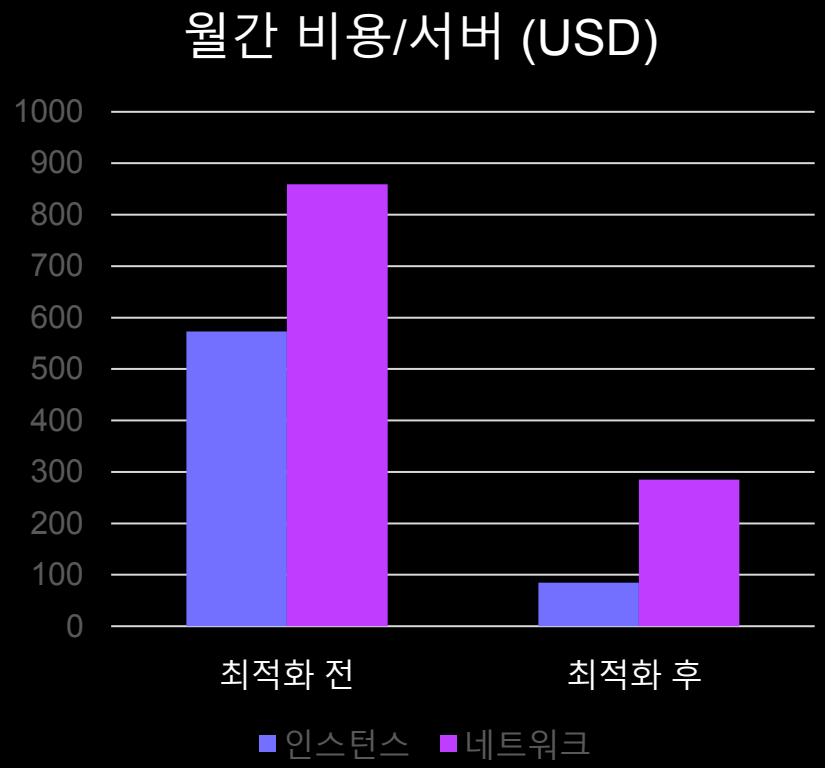
c5.2xlarge[windows]

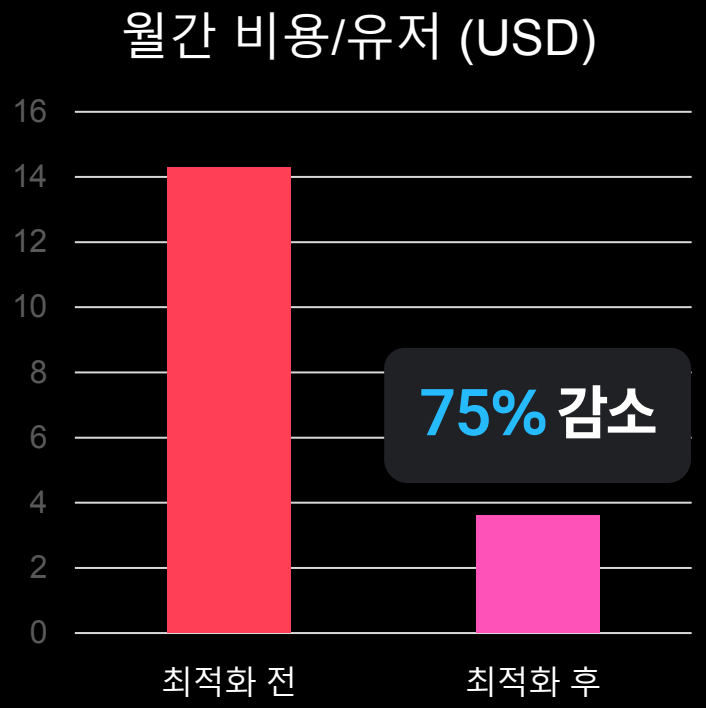
➔ m5.large[linux]



Project-S 최적화(전/후) 비용 추정

유저밀도 1.0

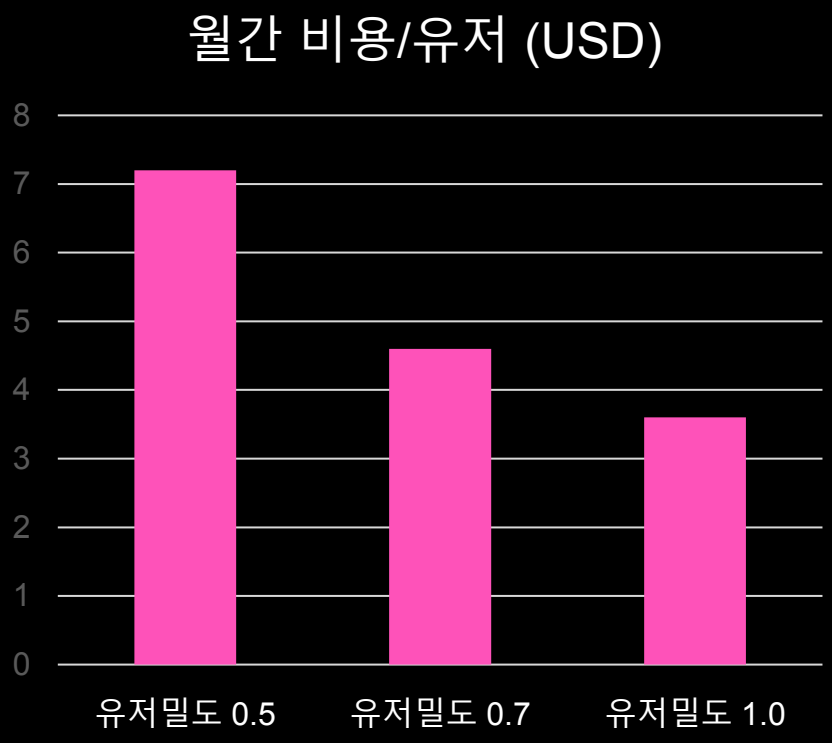




Project-S 최적화(전/후) 비용 추정

같은 조건에도
유저 밀도에 따라 **다른 비용** 발생

가급적 높은 유저밀도 **권장**



Project-S 데디케이티드 서버 비용

월간 비용

13.2억원 ⇨ 2.9억원

서비스 절감 비용

≤10억

최고 동접 7만명
데디케이티드 서버 접속자 수 4.9만명(70%)
유저 밀도 0.5

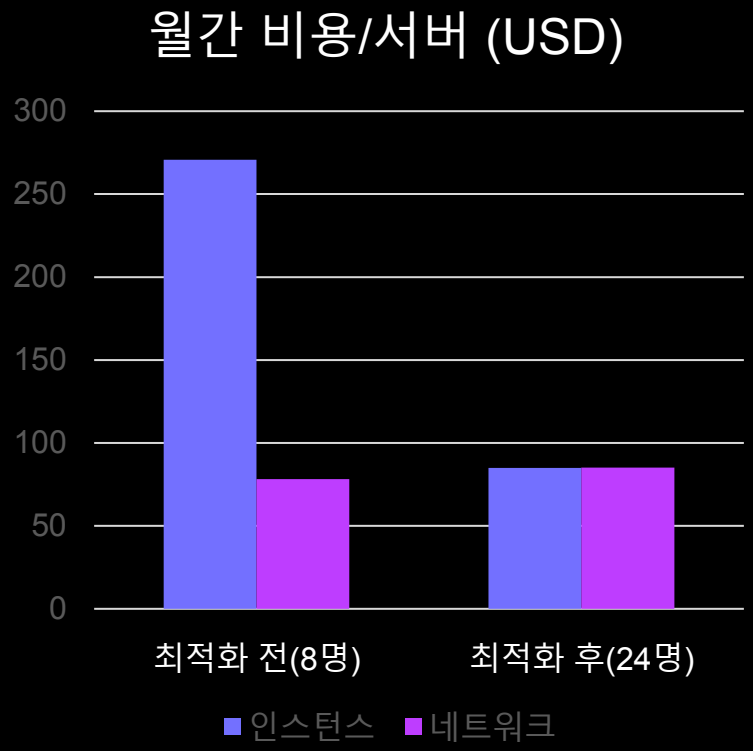
Dragon Sword 월간 비용(예상)

게임세션 1개 ➔ 3개(3x)
플레이어 8명 ➔ 24명(3x)

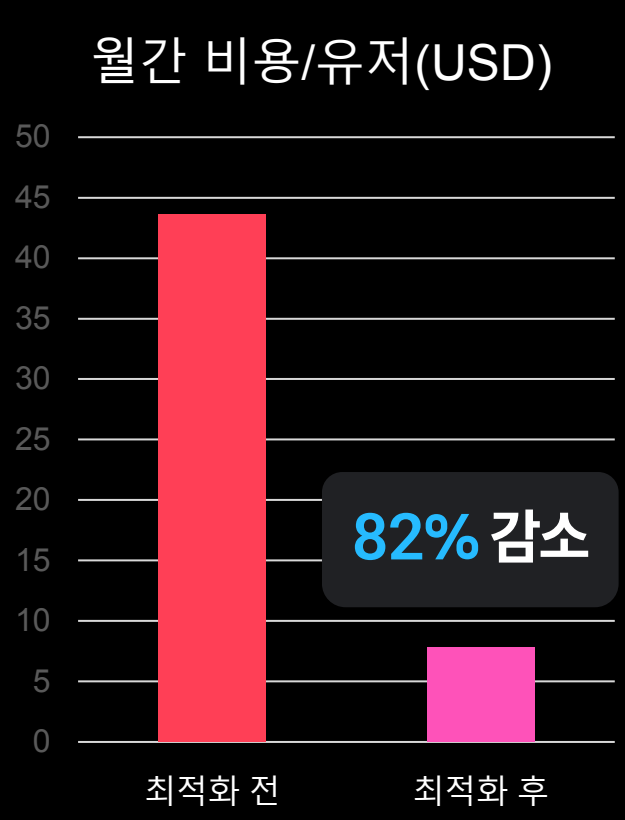
EC2

c5.xlarge(Windows)

➔ m5.large(Linux)

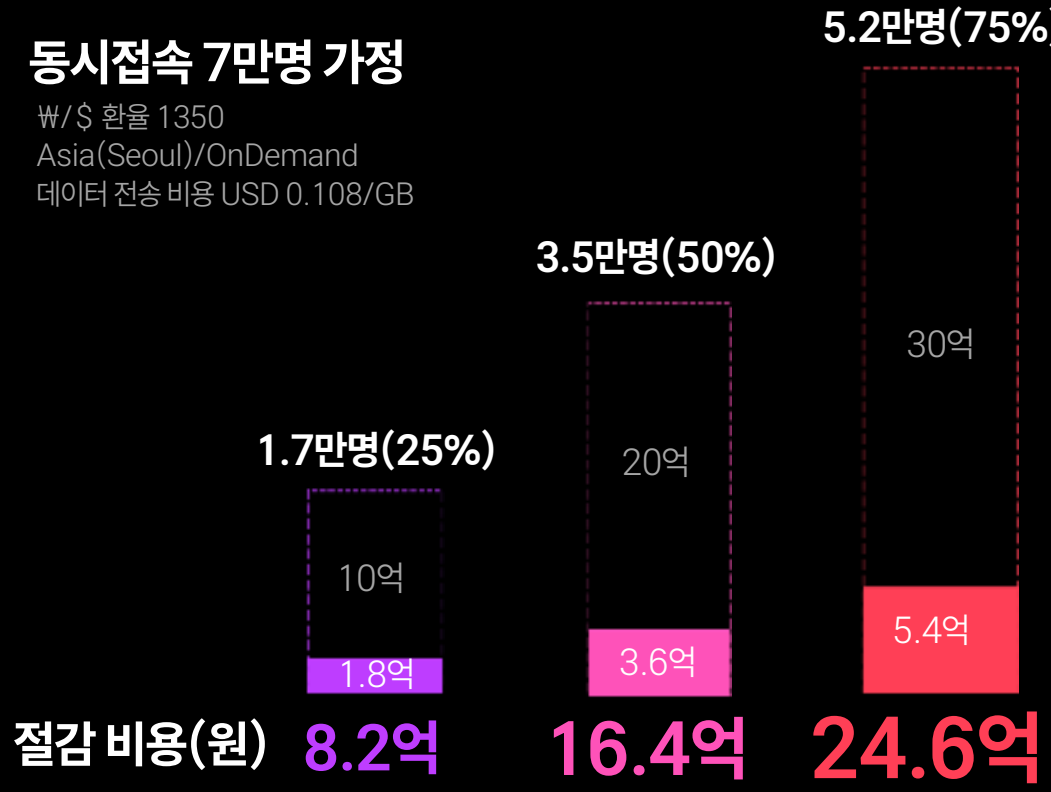


Dragon Sword 월간 비용(예상)



동시접속 7만명 가정

₩/$ 환율 1350
Asia(Seoul)/OnDemand
데이터 전송 비용 USD 0.108/GB




Dragon Sword
전 분야 개발자 채용 중
HOUND 13
Project D

Q&A