Introduzione

Il sistema **KOrche** (*nome temporaneo*) è un *placer*: un servizio che, dato lo stato di un cluster e la descrizione di un nuovo job (o Pod) da collocare, calcola la collocazione ottimale del nuovo carico all'interno del cluster.
A differenza del comportamento standard di **Kubernetes**, questo sistema non si limita a bilanciare le risorse disponibili, ma tiene conto di ulteriori parametri, con particolare attenzione al **costo energetico**, che costituisce lo scopo principale del progetto.

Oltre al consumo energetico, KOrche integra informazioni che Kubernetes non utilizza nativamente, come la **capacità di gestire carichi real-time** e l'**assurance** (affidabilità stimata del nodo).

Il servizio non è un orchestratore attivo o continuo; si tratta di un **servizio su chiamata**: quando si vuole eseguire un nuovo deploy, si effettua una richiesta HTTP POST con un JSON, e KOrche risponde indicando *dove collocare il nuovo job*.

Fornita la risposta, il sistema termina e non viene mantenuto alcuno stato.

Assurance, Criticità e Repliche

In Kubernetes, l'amministratore specifica il numero di repliche desiderato per un servizio (repliche di ridondanza).
KOrche propone un approccio diverso e più probabilistico.

Chi esegue il deploy specifica la **criticità** del job, cioè:

> "La minima probabilità accettabile che almeno la metà delle repliche non subisca interferenze o malfunzionamenti nel cluster."

Il sistema prende questa soglia di *criticità* e cerca un insieme minimo di nodi la cui combinazione di **assurance** soddisfi la probabilità richiesta.

L'**assurance** di un nodo rappresenta la probabilità che un servizio (Pod, Job, carico) deployato su di esso **non subisca interferenze / ritardi** durante la sua esecuzione, considerando anche le condizioni di stress.
Questa metrica dipende dalle caratteristiche Hardware e Software del nodo, ed è immutabile nel tempo.. rappresenta quanto quel nodo è in grado di resistere allo stress, e quanto è in grado di proteggere i propri job da interferenze e ritardi anche in condizioni non ideali.
In laboratorio, abbiamo stimato l'assurance misurando la variazione del tempo di esecuzione del nodo in condizioni di carico e stress diversi. Per ora si possono usare valori stimati o empirici.

Il placer calcola quindi il numero di repliche e i nodi migliori su cui collocarle, cercando di **minimizzare il numero di nodi coinvolti** pur rispettando la criticità richiesta.

Endpoint e invocazione

Il sistema espone due endpoint REST:

`GET /health`

Restituisce un semplice messaggio di stato che indica se il servizio è attivo e pronto a ricevere richieste.
Non richiede alcun body.

`POST /place[?output=file]`

Accetta una richiesta JSON conforme al formato descritto nella sezione *Input / Richiesta* e restituisce un JSON conforme alla sezione *Output / Risposta*.

Per impostazione predefinita, la risposta viene restituita nel body HTTP.

È possibile aggiungere il parametro opzionale:

`?output=file`

per richiedere che il risultato venga salvato come file JSON sul filesystem del servizio.
In tal caso, la risposta HTTP conterrà il percorso del file generato anziché il JSON completo.

Esempio di invocazione:

`POST /place?output=file`

Importante:
KOrche *non è un orchestratore attivo o sempre in esecuzione*.
Non raccoglie metriche, non osserva il cluster in tempo reale, e non effettua il deployment effettivo.
È un **servizio consulente**, chiamato on-demand per decidere *dove collocare un nuovo job*.

Input e Output

Le interazioni avvengono tramite scambio di JSON.

Input – Richiesta

La richiesta descrive:

1. Lo stato corrente del cluster (`cluster`)
2. Il job da collocare e la configurazione dell'algoritmo (`deployment`)

Struttura generale:

```
{
  "cluster": {
    "worker_nodes": [ { ... }, ... ]
  },
  "deployment": {
    "pod": { ... },
    "algorithm": { ... }
  }
}
```

Chiavi principali

Campo	Tipo	Obbligatorio	Descrizione
`cluster`	oggetto	✅	Contiene la chiave `"worker_nodes"`: una lista di specifiche dei nodi del cluster
`deployment.pod`	oggetto	✅	Pod / Job da collocare.
`deployment.algorithm`	oggetto	✅	Parametri e tipo di algoritmo di placement.

Struttura dei Worker Nodes

```
{
  "id": "worker-large-01",
  "cpu": { "capacity": "16", "requested": "0", "limit": "0" },
  "memory": { "capacity": "16Gi", "requested": "0", "limit": "0" },
  "storage": { "capacity": "80gi", "requested": "0", "limit": "0" },
  "real_time": true,
  "energy_cost": 180,
  "energy_cost_idle": 30,
  "assurance": 0.95
 }
```

Ogni `worker_node` è un oggetto con i seguenti campi:

Campo	Tipo	Obbligatorio	Descrizione
`id`	string	✅	Identificativo del nodo
`cpu`, `memory`, `storage`	oggetto	✅	Risorse del nodo (capacità, richieste, limiti)
`real_time`	bool	✅	`true` se il nodo è RT-capable. `false` altrimenti
`energy_cost`	int	✅	Costo energetico massimo del nodo. se `energyCostMode` è `'absolute'` o `'deltaabsolute'` , questo è il costo del nodo se è attivo; se `energyCostMode` è `'linear'` o `'deltalinear'` , questo è il costo quando la cpu è usata al 100%, e viene calcolato linearmente.
`energy_cost_idle`	int	opzionale default: 0	Costo energetico del nodo quando è *Idle* (nessun job deployato). Funge anche da minimo qualora la percentuale di utilizzo moltiplicata per `energy_cost` risulti minore.
`assurance`	float [0–1]	✅	Probabilità di affidabilità

Ogni risorsa (`cpu`, `memory`, `storage`) richiede i campi:

```
{
  "capacity": string,
  "requested": string,
  "limit": string
}
```

Unità di misura:

- **CPU (cores)**: `m` (millicores), o nessuna unità per intendere *cores*
- **Memoria/Storage (bytes)**: `Ki`, `Mi`, `Gi`, `Ti`, `Pi`, `Ei` oppure `k`, `M`, `G`, `T`, `P`, `E`. Oppure notazione scientifica. Laddove non c'è unità si intendono bytes

`In generale è ammesso tutto ciò che è previsto da Kubernetes (resource-units-in-kubernetes)`

Questi tre valori rappresentano:

Campo	Descrizione
`capacity`	**(Solo per i nodi)** Capacità massima del nodo, per quella risorsa. Questo valore non varierà mai e dipende solo dalle caratteristiche del nodo
`requested`	Quantità richiesta allocata e riservata (per i nodi) o da richiedere e riservare (per i Job). Se questo valore dovesse eccedere la capacità, questo nodo verrebbe scartato dalla selezione in quanto non idoneo.
`limit`	Somma delle quantità limite richieste (per i nodi) o quantità limite che il Job dichiara che potrà usare (per i Job). Nei nodi, questo valore può eccedere la capacità, ma ciò vorrebbe dire che il sistema è quasi saturo. Il placer tiene conto di questo e scoraggia questa situazione. Per scoraggiare ancora di più si può ridurre l'iperparametro `overcommit_ReourceDangerRatio` .

Struttura del Pod (deplyment.pod)

```
“pod”: {
  "id": "pod-01",
  "real_time": true,
  "criticality": 0.8,
  "cpu": { "requested": "500m", "limit": "2000m" },
  "memory": { "requested": "2Gi", "limit": "4Gi" },
  "storage": { "requested": "20Gi", "limit": "20Gi" }
}
```

Campo	Tipo	Obbligatorio	Descrizione
`id`	string	✅	Identificativo del pod o job.
`real_time`	bool	✅	Indica se il pod richiede capacità real-time.
`criticality`	float [0–1]	✅	Criticità del job. Indica la probabilità minima accettabile che almeno metà delle repliche non subiscano interferenze.
`cpu`	oggetto	✅	Risorse CPU richieste e limite massimo consentito.
`memory`	oggetto	✅	Risorse di memoria richieste e limite massimo.
`storage`	oggetto	✅	Spazio di archiviazione richiesto e limite massimo.

Ogni risorsa (`cpu`, `memory`, `storage`) ha la stessa struttura mostrata in `worker_node` (esclusa la capacità):

Struttura di algorithm (deployment.algorithm)

L’oggetto `algorithm` definisce il tipo di strategia di placement da usare, i dettagli della funzione multi-obiettivo, e eventuali iperparamteri.

```
"algorithm": {
    "type": "DP_StateAware",
    "resourceFit": "LeastAllocated",
    "weights": {
      "resourceFit": 3,
      "energyCost": 2,
      "assuranceWasteless": 1,
      "rtWasteless": 1,
      "limitOvercommitPenalty" : 1
    },
    "hyperparams": {
      "maxReplicas": 3,
      "dp_maxNeighbors": 10,
      "dp_solutionOversizeSearch" : 1,
      "dp_neighborSpan" : 1,
      "dp_energyCostWakeupMultiplier" : 1.5,
      "dp_scoreAggregationMode" : "SquaredSum",
      "overcommit_ReourceDangerRatio" : 0.9,
      "energyCostMode" : "deltalinear",
      "verbose" : true
    }
  }
```

Campo	Tipo	Obbligatorio	Descrizione
`type`	string	✅	Tipo di algoritmo di placement da utilizzare. Valori ammessi: `"Greedy"`, `"DP_StateAware"`, `"DP_StateAgnostic"`, `"K8s"`.
`resourceFit`	string	consigliato	Tipo di funzione di Resource Fit da usare. Valori ammessi: `"LeastAllocated"` (default sia nostro che di Kubernetes), `"MostAllocated"`, `"RequestedToCapacityRatio"`.
`weights`	mappa (string→float)	opzionale	Pesi relativi dei diversi obiettivi di scoring. Se assenti, viene posto `resourceFit` a 1 e tutti gli altri a 0.

`hyperparams`	oggetto	opzionale	Iperparametri che controllano il comportamento dell'algoritmo (in particolare i metodi DP).

Algoritmi di Placement

I quattro algoritmi disponibili sono alternativi, ma condividono interfaccia e struttura logica.
Tutti cercano di soddisfare il vincolo di criticità richiesto dal Pod, basandosi sulla funzione multi-obiettivo fornita.

1. K8s

Presente solo per confronto. In teoria si comporta come Kubernetes ma utilizzando la funzione multi-obiettivo fornita. Inoltre, continua ad aggiungere nodi finché non soddisfa il vincolo di criticità (cosa che Kubernetes non gestisce di suo)

È presente solo come default, ma non c'è reale motivo di utilizzarla.

---

2. Greedy

L'algoritmo **Greedy** procede in modo iterativo, inseguendo l'ottimo locale ad ogni scelta.
A ogni passo seleziona il nodo con score minimo rispetto al pod e lo aggiunge alla soluzione finché la criticità richiesta non è soddisfatta.

Originariamente non era altro che K8s, con la funzione multi-obiettivo; ora, in aggiunta, separa il set di nodi in **già attivi** e **idle** ed esegue il confronto inizialmente solo sui nodi attivi, così da ridurre il delta energetico.

Se non riesce a completare la soluzione in questo modo, passa ad analizzare i nodi *Idle*.

- È l'algoritmo più veloce in termini di complessità computazionale.
- Può tuttavia non restituire la soluzione globalmente ottima.

---

3. DP_StateAware

Algoritmo basato su **programmazione dinamica**.
Come Greedy, considera inizialmente solo i nodi attivi e, se necessario, estende la ricerca anche agli idle.

- Costruisce una matrice che mappa, per diverse combinazioni di nodi, il *vincolo di Criticità* alla ricerca di un primo gruppo di soluzioni.

- Partendo da queste, esplora intorni vicini per migliorare lo score complessivo.
- Include parametri di controllo (`dp_maxNeighbors`, `dp_neighborSpan`, ecc.) per limitare la complessità computazionale.
- Quando considera contemporaneamente nodi *Attivi* ed *Idle* fornisce un sovraccosto per questi ultimi (a differenza di Greedy che non li analizza mai assieme)

---

4. DP_StateAgnostic

Variante del precedente che considera fin dall'inizio sia nodi attivi che idle.

- Negli esperimenti ha fornito le prestazioni migliori, ma è anche il più oneroso in termini di complessità computazionale.

N.B.: Questo maggiore costo computazionale si avverte per cluster medio-grandi (25+ nodi), al di sotto l'overhead è abbastanza trascurabile

---

Funzioni di Scoring

Ogni algoritmo usa una **funzione di scoring multi-obiettivo** per valutare la bontà di una coppia (nodo, pod).
Lo score è una *funzione costo*: **più basso è, meglio è**.
La funzione aggrega più obiettivi con una **somma pesata normalizzata**.

Le componenti sono le seguenti:

Resource Fit

La funzione di **Resource Fit** misura quanto un nodo è "pieno" rispetto alla sua capacità disponibile, in base al rapporto tra risorse richieste e capacità totale.
È la stessa logica di base usata da Kubernetes, ma con tre varianti selezionabili.

Va specificata nel campo ***deployment.algorithm.resourceFit***

Variante	Significato
`LeastAllocated`	**Privilegia i nodi meno saturi**: bilancia il carico distribuendo equamente, ma è la più onerosa in termini di *costo energetico*. È la modalità predefinita di KOrche.

`MostAllocated`	**Privilegia i nodi già carichi**: tende a concentrare i pod dove c'è già attività, riducendo la dispersione e il numero di nodi attivi (utile per minimizzare il consumo energetico).
`RequestedToCapacityRatio`	**Minimizza lo spazio inutilizzato sui nodi.** Variante più dinamica, utilizzata da Kubernetes per ottimizzare la percentuale d'uso delle risorse, bilanciando saturazione e frammentazione.

Energy Cost

Penalizza i nodi con maggiore `energy_cost`.
Il comportamento di questa funzione dipende dall' iper-parametro '`energyCostMode`'

absolute : Funzione gradino. Il costo del nodo è *EnergyCost_Idle* se il nodo è Idle, *EnergyCost* se il nodo è attivo

delta : Funzione gradino. come 'absolute' ma lo score è calcolato in base alla differenza apportata al costo del cluster e non in base al costo del nodo in quanto tale.
La differenza in termini di risultato è che, tra nodi già attivi, 'absolute' preferisce nodi a basso costo mentre 'delta' è indifferente.

linear : Funzione lineare dipendente dall'attuale carico della cpu (*p*). Permette di rappresentare il costo energetico in maniera più dinamica.

deltaLinear: Unione delle due. Calcola il costo in maniera lineare sul carico della CPU, e lo score in base alla differenza apportata al costo totale del cluster.
In questo caso, nella scelta tra due nodi già attivi, questa funzione prediligerà nodi il cui costo energetico cresce più lentamente.

Assurance Wasteless e RT Wasteless

Penalizzano l'uso inefficiente di nodi "speciali":

- **RT Wasteless**: Scoraggia la selezione di nodi con 'real_time : true' per job non real-time.
  Questo serve a preservare queste risorse considerate più preziose.

- **Assurance Wasteless**: preferisce nodi con assurance prossima alla criticità. Ed equipara i nodi le cui assurances soddisfano automaticamente la criticità.
  Scoraggia la selezione di nodi con assurance bassa, i quali richiederebbero più repliche o potrebbero essere più rischiosi; allo stesso tempo, evita di fornire sempre i nodi con miglior assurance possibile, per non restare senza in caso di necessità.

Limit Overcommit

Penalizza situazioni in cui la somma dei limiti di risorsa supera la capacità del nodo.

Kubernetes standard ignora questo tipo di vincolo, ma in sistemi più sensibili (es. real-time o energy-aware) l'overcommitment può aumentare la probabilità di interferenze.
Questa funzione valuta il rischio di "sforamento" in modo graduale e continuo.

Questa funzione è parametrizzabile tramite l'iperparametro '`overcommit_ReourceDangerRatio`' che indica la percentuale della capacity che, se sforata dalla somma dei limits, deve causare uno scoraggiamento da parte della funzione.

Pesi (campo deployment.algorithm.weights)

Ogni chiave rappresenta una sottofunzione di scoring.
I pesi sono normalizzati automaticamente, quindi conta il rapporto relativo e non il valore assoluto.

Chiave	Significato
`resourceFit`	Bilanciamento delle risorse (funzione principale).
`energyCost`	Penalizza i nodi con costo energetico maggiore.
`assuranceWasteless`	Penalizza l'uso inefficiente di nodi con alta assurance.
`rtWasteless`	Penalizza l'uso di nodi RT per job non RT.
`limitOvercommitPenalty`	Penalizza il superamento dei limiti di risorsa.

Iperparametri (campo deployment.algorithm.hyperparams)

Controllano il comportamento degli algoritmi, in particolare quelli basati su programmazione dinamica (DP).

Sono tutti opzionali.

Chiave	Tipo	Default	Descrizione
`maxReplicas`	int	-1	Numero massimo di repliche per job. -1: Nessun limite
`dp_maxNeighbors`	int	10	Numero massimo di soluzioni vicine da esplorare. -1: Nessun limite
`dp_solutionOversizeSearch`	int	2	Numero massimo di repliche aggiuntive oltre la soluzione minima accettata.
`dp_neighborSpan`	int	2	Ampiezza della ricerca degli intorni. Aumenta notevolmente il costo computazionale.
`dp_energyCostWakeupMultiplier`	float	1.5	Moltiplicatore del costo energetico per nodi inattivi (idle).
`dp_scoreAggregationMode`	string	`"SquaredSum"`	Metodo di aggregazione degli score per le funzioni DP: `"Sum"`, `"Geometric"`, `"SquaredSum"`.
`overcommit_ReourceDangerRatio`	float	`0.95`	Percentuale delle *capacities* delle risorse dalla quale in poi iniziare a scoraggiare quel nodo. 0.95 vuol dire che se CPU.limit supererebbe il 95% di CPU.Capacity, allora la funzione `limitOvercommitPenalty` deve scoraggiare questa scelta.
`verbose`	bool	false	Rende più dettagliata la stringa `'explanation'` nella risposta

`energyCostMode`	string	`"absolute"`	Modalità di calcolo del costo energetico. `absolute`: Funzione gradino.. 0 se il nodo è *Idle*, *EnergyCost* altrimenti. `linear`: Crescità lineare in funzione della percentuale di carico sulla CPU. `delta`: Rapporto sulla differenza di EnergyCost totale del cluster.. resitutisce 0 se il nodo è già in uso. `deltaLinear`: Come sopra, ma in rapporto alla differenza di Energy Cost

---

Struttura della Risposta (Output)

L'output del servizio è un oggetto JSON che rappresenta la **decisione di placement** del sistema.
Contiene l'esito della valutazione, il numero di repliche consigliato e i dettagli sui nodi scelti per l'allocazione.

Il servizio **non effettua direttamente il deploy**, ma suggerisce la migliore configurazione di posizionamento sulla base dello stato corrente del cluster e dei vincoli del Pod.

Struttura generale

Campo	Tipo	Descrizione
`accepted`	boolean	Indica se la richiesta è stata accettata (`true`) o rifiutata (`false`).
`replicas`	int	Numero di repliche del Pod richieste per soddisfare la criticità.
`probability`	float	Probabilità stimata che almeno metà delle repliche non subiscano interferenze (valore tra 0 e 1). Controparte del vincolo di criticità.

`deploy_on`	array di oggetti	Elenco dei nodi selezionati per l'allocazione del Pod. Ogni oggetto include l'ID del nodo, le sue risorse aggiornate e i punteggi di valutazione.
`explanation`	string	Descrizione testuale dell'esito, utile per debugging o tracciamento decisionale.
`energy_delta`	float	Differenza di costo energetico del cluster prima e dopo il placement. Positivo se l'allocazione prevede nodi originariamente *Idle*.

Struttura del campo `deploy_on`

Ogni elemento della lista `deploy_on` rappresenta un nodo selezionato per ospitare una replica del Pod.

Campo	Tipo	Descrizione
`id`	string	Identificativo del nodo.
`cpu`	oggetto	Stato aggiornato della CPU del nodo (dopo l'allocazione del Pod).
`memory`	oggetto	Stato aggiornato della memoria.
`storage`	oggetto	Stato aggiornato dello storage.
`score`	float	Punteggio complessivo assegnato alla coppia nodo–pod. Più basso è, meglio è.
`explain`	mappa (string→float)	Punteggi parziali per ciascuna funzione obiettivo (come `resourceFit`, `energyCost`, ecc.).

Esempio di response:


```
{
  "accepted": true,
  "replicas": 1,
  "probability": 0.95,
  "deploy_on": [
    {
      "id": "worker-large-02",
      "cpu": {
        "capacity": "16000m",
        "requested": "9000m",
        "limit": "14000m"
      },
      "memory": {
        "capacity": "16Gi",
        "requested": "8Gi",
        "limit": "11Gi"
      },
      "storage": {
        "capacity": "100Gi",
        "requested": "90Gi",
        "limit": "143Gi"
      },
      "score": 0.15174508,
      "explain": {
        "assuranceWasteless": 0,
        "energyCost": 0.02578125,
        "limitOvercommitPenalty": 0.041588824,
        "resourcesFit": 0.084375,
        "rtWasteless": 0
      }
    }
  ],
  "explanation": "1-replica placement suggested to grant an 80%
affidability.",
  "energy_delta": 61.875
}
```