Red Hat Ansible Automation Platform 2.4

Operator ベースのインストール向けの Red Hat Ansible Automation Platform のパフォーマンスに関する考慮事項

Operator ベースのインストールでのパフォーマンスを向上させるために Automation Controller を設定する

Last Updated: 2024-06-18

Red Hat Ansible Automation Platform 2.4 Operator ベースのインストール向けの Red Hat Ansible Automation Platform のパフォーマンスに関する考慮事項

Operator ベースのインストールでのパフォーマンスを向上させるために Automation Controller を設定する

法律上の通知

概要

このガイドでは、Automation Controller とコンテナーグループのリソース要求、およびその他の Kubernates 設定オプションを設定して、Automation Controller の Operator ベースのインストールで大規模かつ効率的にジョブを実行するための推奨事項を提供します。

目次

はじめに

アプリケーションを Red Hat OpenShift Container Platform などのコンテナーオーケストレーションプラットフォームにデプロイすると、運用上の観点から多くの利点を得ることができます。たとえば、アプリケーションのベースイメージの更新は、単純なインプレースアップグレードで、中断をほとんど、または一切発生させずに実行できます。従来の仮想マシンにデプロイされたアプリケーションの必須オペレーティングシステムをアップグレードすることは、中断の可能性とリスクがはるかに高いプロセスになる可能性があります。

アプリケーションおよび Operator の開発者は、アプリケーションのデプロイメントにおいて、OpenShift Container Platform ユーザーに多くのオプションを提供できますが、これらの設定は OpenShift Container Platform の設定に依存しているため、エンドユーザーが提供する必要があります。

たとえば、Openshift クラスターでノードラベルを使用すると、特定のノードでさまざまなワークロードが確実に実行できるようになります。Ansible Automation Platform Operator にはこれを推測する手段がないため、このタイプの設定はユーザーが提供する必要があります。

RED HAT ドキュメントへのフィードバック (英語のみ)

このドキュメントを改善するための提案がある場合、またはエラーを見つけた場合は、テクニカルサポート (https://access.redhat.com) に連絡し、**docs-product** コンポーネントを使用して Ansible Automation Platform Jira プロジェクトで Issue を作成してください。

第1章 POD の仕様変更

1.1. 概要

Pod の Kubernetes の概念は、「1 つのホストに一緒にデプロイされる 1 つ以上のコンテナーであり、定義、デプロイ、または管理できる最小のコンピューティングユニット」です。

Pod は、コンテナーに対して、(物理または仮想) マシンインスタンスと同等のものです。各 Pod は独自の内部 IP アドレスで割り当てられるため、そのポートスペース全体を所有し、Pod 内のコンテナーはそれらのローカルストレージおよびネットワークを共有できます。

Pod にはライフサイクルがあります。それらは定義された後にノードで実行するために割り当てられ、コンテナーが終了するまで実行するか、その他の理由でコンテナーが削除されるまで実行します。ポリシーおよび終了コードによっては、Pod は終了後に削除されるか、コンテナーのログへのアクセスを有効にするために保持される可能性があります。

Red Hat Ansible Automation Platform は単純なデフォルトの Pod 仕様を提供しますが、デフォルトの Pod 仕様をオーバーライドするカスタム YAML または JSON ドキュメントを提供できます。このカスタムドキュメントは、有効な Pod JSON または YAML としてシリアル化できる **ImagePullSecrets** などのカスタムフィールドを使用します。

オプションの完全なリストは、Openshift Online のドキュメント記載されています。

長期サービスを提供する Pod の例。

この例は、数多くの Pod 機能を示していますが、そのほとんどは他のトピックで説明されるため、ここでは簡単に説明します。

```
apiVersion: v1
kind: Pod
metadata:
  annotations: { ... }                    (1)
  labels:
    deployment: docker-registry-1
    deploymentconfig: docker-registry
    docker-registry: default
  generateName: docker-registry-1-        (2)
spec:
  containers:                             (3)
  - env:                                  (4)
    - name: OPENSHIFT_CA_DATA
      value: ...
    - name: OPENSHIFT_CERT_DATA
      value: ...
    - name: OPENSHIFT_INSECURE
      value: "false"
    - name: OPENSHIFT_KEY_DATA
      value: ...
    - name: OPENSHIFT_MASTER
      value: https://master.example.com:8443
    image: openshift/origin-docker-registry:v0.6.2  (5)
    imagePullPolicy: IfNotPresent
    name: registry
    ports:                                (6)
```

```
    - containerPort: 5000
      protocol: TCP
    resources: {}                          7
    securityContext: { ... }               8
    volumeMounts:                          9
    - mountPath: /registry
      name: registry-storage
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: default-token-br6yz
      readOnly: true
  dnsPolicy: ClusterFirst
  imagePullSecrets:                        10
  - name: default-dockercfg-at06w
  restartPolicy: Always                    11
  serviceAccount: default                  12
  volumes:                                 13
  - emptyDir: {}
    name: registry-storage
  - name: default-token-br6yz
    secret:
      secretName: default-token-br6yz
```

ラベル	説明
annotations:	Pod には 1 つまたは複数のラベルでタグ付けすることができ、このラベルを使用すると、一度の操作で Pod グループの選択や管理が可能になります。これらのラベルは、キー/値形式で metadata ハッシュに保存されます。この例で使用されているラベルは **docker-registry=default** です。
generateName:	Pod にはそれらの名前空間内に任意の名前がなければなりません。Pod 定義では、**generateName** 属性を使用して名前のベースを指定し、ランダムな文字を自動的に追加して一意の名前を生成できます。
containers:	**containers** は、コンテナー定義のアレイを指定します。(ほとんどの場合と同様に) この場合は、コンテナーを 1 つだけ定義します。
env:	環境変数は、必要な値を各コンテナーに渡します。
image:	Pod の各コンテナーは、独自の Docker 形式のコンテナーイメージからインスタンス化されます。
ports:	コンテナーは、Pod の IP で使用可能になったポートにバインドできます。
resources:	Pod を指定する場合は、必要に応じて、コンテナーが必要とする各リソースの量を記述できます。指定する最も一般的なリソースは、CPU とメモリー (RAM) です。他のリソースも使用できます。
securityContext:	OpenShift Online は、コンテナーが特権付きコンテナーとして実行を許可されるか、選択したユーザーとして実行を許可されるかどうかを指定するセキュリティーコンテキストを定義します。デフォルトのコンテキストには多くの制限がありますが、管理者は必要に応じてこれを変更できます。

ラベル	説明
volumeMounts:	コンテナーは外部ストレージボリュームがコンテナー内にマウントされるかどうかを指定します。この場合、レジストリーのデータを保存するためのボリュームと、OpenShift Online API に対して要求を行うためにレジストリーが必要とする認証情報へのアクセス用のボリュームがあります。
ImagePullSecrets	Pod には 1 つ以上のコンテナーを含めることができます。コンテナーは、レジストリーからプルする必要があります。認証を必要とするレジストリーからのコンテナーの場合は、名前空間に存在する **ImagePullSecrets** を参照する **ImagePullSecrets** のリストを提供できます。これらを指定すると、イメージをプルする際に Red Hat OpenShift Container Platform がコンテナーレジストリーで認証できるようになります。詳細は、Kubernetes ドキュメントの Resource Management for Pods and Containers を参照してください。
restartPolicy:	Pod 再起動ポリシーと使用可能な値の **Always**、**OnFailure**、および **Never** です。デフォルト値は **Always** です。
serviceAccount:	OpenShift Online API に対して要求する Pod は一般的なパターンです。この場合は **serviceAccount** フィールドがありますが、これは要求を行う際に Pod が認証する必要のあるサービスアカウントユーザーを指定するために使用されます。これにより、カスタムインフラストラクチャーコンポーネントの詳細なアクセス制御が可能になります。
volumes:	Pod は、コンテナーで使用できるストレージボリュームを定義します。この場合は、レジストリーストレージの一時的なボリュームおよびサービスアカウントの認証情報が含まれる secret ボリュームが提供されます。

Automation Controller を使用し、Automation ControllerUI で Pod 仕様を編集し、Kubernetes ベースのクラスターでジョブを実行するために使用される Pod を変更できます。ジョブを実行する Pod の作成に使用される Pod 仕様は YAML 形式です。Pod 仕様の編集の詳細は、Pod 仕様のカスタマイズ を参照してください。

1.1.1. Pod 仕様のカスタマイズ

次の手順で Pod をカスタマイズできます。

手順

1. Automation Controller UI で、**Administration → Instance Groups** に移動します。
2. **Customize pod specification** にチェックを入れます。
3. **Pod Spec Override** フィールドで、トグルを使用して名前空間を指定し、Pod Spec **Pod Spec Override** フィールドを有効にして展開します。
4. **Save** をクリックします。
5. オプション: 追加でカスタマイズする場合は、**Expand** をクリックしてカスタマイズウィンドウ全体を表示します。

ジョブの起動時に使用されるイメージは、ジョブに関連付けられた実行環境によって決まります。Container Registry 認証情報が実行環境に関連付けられている場合、Automation Controller は **ImagePullSecret** を使用してイメージをプルします。シークレットを管理するパーミッションをサービスアカウントに付与しない場合は、**ImagePullSecret** を事前に作成してそれを Pod 仕様で指定し、使用する実行環境から認証情報を削除する必要があります。

1.1.2. Pod による他のセキュアなレジストリーからのイメージ参照の許可

コンテナーグループが、認証情報を必要とするセキュアなレジストリーのコンテナーを使用する場合は、コンテナーレジストリーの認証情報を、ジョブテンプレートに割り当てられている実行環境に関連付けることができます。Automation Controller はこれを使用して、コンテナーグループジョブが実行される OpenShift Container Platform の名前空間に **ImagePullSecret** を作成し、ジョブの完了後にクリーンアップします。

ContainerGroup の名前空間に **ImagePullSecret** がすでに存在する場合は、ContainerGroup のカスタム Pod 仕様で **ImagePullSecret** を指定できます。

コンテナーグループで実行しているジョブで使用されるイメージは、必ずそのジョブに関連付けられている実行環境によってオーバーライドされることに注意してください。

事前に作成された ImagePullSecrets の使用 (詳細)

このワークフローを使用して **ImagePullSecret** を事前に作成する場合は、以前にセキュアなコンテナーレジストリーにアクセスしたシステム上のローカル **.dockercfg** ファイルから作成に必要な情報を入手できます。

手順

.dockercfg file ファイル (新しい Docker クライアントの場合は **$HOME/.docker/config.json**) は、ユーザーの情報を保管する Docker 認証情報ファイルです (以前にセキュアな/セキュアではないレジストリーにログインしている場合)。

1. セキュアなレジストリーの **.dockercfg** ファイルがすでにある場合は、次のコマンドを実行して、そのファイルからシークレットを作成できます。

```
$ oc create secret generic <pull_secret_name> \
--from-file=.dockercfg=<path/to/.dockercfg> \
--type=kubernetes.io/dockercfg
```

2. または、**$HOME/.docker/config.json** ファイルがある場合は、以下のコマンドを実行します。

```
$ oc create secret generic <pull_secret_name> \
--from-file=.dockerconfigjson=<path/to/.docker/config.json> \
--type=kubernetes.io/dockerconfigjson
```

3. セキュアなレジストリーの Docker 認証情報ファイルがまだない場合は、次のコマンドを実行してシークレットを作成できます。

```
$ oc create secret docker-registry <pull_secret_name> \
--docker-server=<registry_server> \
--docker-username=<user_name> \
--docker-password=<password> \
--docker-email=<email>
```

4. Pod のイメージをプルするためにシークレットを使用するには、サービスアカウントにシークレットを追加する必要があります。この例では、サービスアカウントの名前は、Pod が使用するサービスアカウントの名前に一致している必要があります。デフォルトはデフォルトのサービスアカウントです。

```
$ oc secrets link default <pull_secret_name> --for=pull
```

5. オプション: ビルドイメージのプッシュおよびプルにシークレットを使用するには、Pod 内にシークレットがマウント可能でなければなりません。以下でこれを実行できます。

```
$ oc secrets link builder <pull_secret_name>
```

6. オプション: ビルドについては、ビルド設定内からのプルシークレットとしてシークレットを参照する必要もあります。

コンテナーグループが正常に作成されると、新しく作成されたコンテナーグループの **Details** タブが表示されます。これにより、コンテナーグループ情報を確認し、編集できます。これは、**インスタンスグループ** のリンクから、**Edit** アイコン ✎ をクリックした場合に開くメニューと同じです。インスタンスを編集し、このインスタンスグループに関連付けられたジョブを確認することもできます。

1.2. POD とコンテナーのリソース管理

Pod を指定すると、コンテナーが必要とする各リソースの量を指定できます。指定する最も一般的なリソースは、CPU とメモリー (RAM) です。

Pod 内のコンテナーのリソース要求を指定すると、kubenetes-scheduler はこの情報を使用して、Pod を配置するノードを割り当てます。

コンテナーのリソース制限を指定すると、**kubelet** またはノードエージェントがその制限を適用し、実行中のコンテナーが設定した制限を超えてそのリソースを使用することを許可しないようにします。また、kubelet は、少なくとも要求された量のシステムリソースを、そのコンテナーが使用するために予約します。

1.2.1. 要求と制限

Pod が実行されているノードに使用可能なリソースが十分にある場合、コンテナーは、そのリソースに対する要求が指定するよりも多くのリソースを使用する可能性があります。ただし、コンテナーはそのリソース制限を超えて使用することはできません。

たとえば、コンテナーに 256 MiB のメモリー要求を設定し、そのコンテナーが 8 GiB のメモリーを持つノードにスケジュールされた Pod 内にあり、他の Pod がない場合、コンテナーはより多くの RAM を使用しようとする可能性があります。

そのコンテナーに 4 GiB のメモリー制限を設定すると、kubelet とコンテナーのランタイムによって制限が適用されます。ランタイムは、コンテナーが設定されたリソース制限を超えて使用することを防ぎます。

コンテナー内のプロセスが許可された量を超えるメモリーを消費しようとすると、システムカーネルは **Out Of Memory**(OOM) エラーで、割り当てを試みたプロセスを終了します。

制限は 2 つの方法で実装できます。

- 事後対応: 違反が検出されると、システムが介入します。
- 強制: システムは、コンテナーが制限を超えないようにします。

ランタイムが異なれば、同じ制限を実装する方法も異なる場合があります。

注記

リソースの制限を指定しても、要求を指定せず、admission-time メカニズムがそのリソースのデフォルト要求を適用していない場合、Kubernetes は指定された制限をコピーし、それをリソースの要求値として使用します。

1.2.2. リソースタイプ

CPU とメモリーはどちらもリソースのタイプです。リソースタイプには基本の単位があります。CPU は計算処理を表し、Kubernetes CPU の単位で指定されます。メモリーはバイト単位で指定されます。

CPU とメモリーは、まとめてコンピュートリソースまたはリソースと呼ばれます。コンピュートリソースは、要求、割り当て、消費でき、さらに数量を測定できるリソースです。これらは API リソースとは異なります。Pod やサービスなどの API リソースは、Kubernetes API サーバーを介して読み取りおよび変更できるオブジェクトです。

1.2.3. Pod とコンテナーのリソース要求および制限の指定

コンテナーごとに、次のようなリソースの制限と要求を指定できます。

```
spec.containers[].resources.limits.cpu
spec.containers[].resources.limits.memory
spec.containers[].resources.requests.cpu
spec.containers[].resources.requests.memory
```

個々のコンテナーに対してのみ要求と制限を指定できますが、Pod 全体のリソース要求と制限を考慮することも役立ちます。特定のリソースの場合には、Pod リソースの要求または制限は、Pod 内にあるコンテナーごとに割り当てられた、対象タイプのリソース要求または制限を合計したものです。

1.2.4. Kubernetes のリソース単位

CPU のリソース単位

CPU リソースの制限と要求は、CPU 単位で測定されます。Kubernetes における 1 CPU は、ノードが物理ホストか物理マシン内で実行されている仮想マシンかに応じて、1 つの物理プロセッサーコアまたは 1 つの仮想コアに相当します。

部分的な要求は許可されます。**spec.containers[].resources.requests.cpu** を **0.5** に設定してコンテナーを定義すると、1.0 CPU を要求した場合と比較して半分の CPU 時間を要求します。CPU のリソース単位で 0.1 は 100m と同じで、100 millicpu または 100 millicore です。millicpu と millicores は同じ意味です。CPU リソースは常にリソースの絶対量として指定され、相対量として指定されることはありません。たとえば 500m CPU は、そのコンテナーがシングルコア、デュアルコア、または 48 コアのマシンで実行されているかどうかにかかわらず、同じ量のコンピューティングパワーを表します。

注記

1.0 または 1000m 未満の CPU 単位を指定するには、milliCPU 形式を使用する必要があります。たとえば、0.005 CPU ではなく 5m を使用します。

メモリーのリソース単位

メモリーの制限と要求はバイト単位で測定されます。数量を示す E、P、T、G、M、k のいずれかの接尾辞を使用して、メモリーを単純な整数または固定小数点数として表すことができます。Ei、Pi、Ti、Gi、Mi、Ki の 2 のべき乗も使用できます。たとえば、次はほぼ同じ値を表します。

```
128974848, 129e6, 129M,  128974848000m, 123Mi
```

接尾辞の大文字と小文字に注意してください。400m のメモリーを要求する場合、これは 0.4 バイトの要求であり、400 メビバイト (400Mi) でも 400 メガバイト (400M) でもありません。

CPU とメモリーの仕様例

次のクラスターには、専用の 100m CPU と 250Mi でタスク Pod をスケジュールするのに十分な空きリソースがあります。クラスターは、最大 2000m CPU と 2Gi メモリーを専用に使用するバーストにも耐えることができます。

```
spec:
  task_resource_requirements:
    requests:
      cpu: 100m
      memory: 250Mi
    limits:
      cpu: 2000m
      memory: 2Gi
```

Automation Controller は、設定された制限を超えてリソースを使用するジョブをスケジュールしません。タスク Pod が設定された制限を超えてリソースを使用する場合、コンテナーは kubernetes によって OOMKilled され、再起動されます。

1.2.5. リソース要求の推奨サイズ

コンテナーグループを使用するすべてのジョブは、同じ Pod 仕様を使用します。Pod 仕様には、ジョブを実行する Pod のリソース要求が含まれています。

すべてのジョブは同じリソース要求を使用します。Pod 仕様で特定のジョブに指定されたリソース要求は、Kubernetes がワーカーノードで使用可能なリソースに基づいてジョブ Pod をスケジュールする方法に影響します。これらはデフォルト値です。

- 通常、1 つのフォークには 100 Mb のメモリーが必要です。これは **system_task_forks_mem** を使用して設定されます。ジョブに 5 つのフォークがある場合、ジョブ Pod の仕様は 500 Mb のメモリーを要求する必要があります。
- フォークの値が特に高いジョブテンプレートや、より大きなリソース要求が必要なジョブテンプレートでは、より大きなリソース要求を示す別の Pod 仕様で別のコンテナーグループを作成する必要があります。その後、それをジョブテンプレートに割り当てることができます。たとえば、フォーク値が 50 のジョブテンプレートは、5 GB のメモリーを要求するコンテナーグループとペアにする必要があります。
- ジョブのフォーク値が高く、単一 Pod にそのジョブを含めることができない場合は、ジョブスライス機能を使用します。これにより、コンテナーグループによってプロビジョニングされた自動化 Pod に個々のジョブスライスが収まるように、インベントリーが分割されます。

第2章 コントロールプレーンの調整

コントロールプレーンとは、ユーザーインターフェイスなどを提供し、ジョブのスケジューリングと起動を処理する Web コンテナーとタスクコンテナーを含む Automation Controller Pod を指します。Automation Controller カスタムリソースでは、**レプリカ** の数によって、Automation Controller デプロイメント内の Automation Controller Pod の数が決まります。

2.1. タスクコンテナーの要求と制限

タスクコンテナーの CPU とメモリーのリソース制限値を設定する必要があります。実行ノードで実行されるジョブごとに、そのジョブのコールバックイベントをスケジュール、起動、および受信する処理をコントロールプレーンで行う必要があります。

Automation Controller の Operator デプロイメントの場合、このコントロールプレーンの使用容量は、**controlplane** インスタンスグループで追跡されます。使用可能な容量は、Automation Controller 作成時に Automation Controller の仕様または OpenShift UI の **task_resource_requirements** フィールドで、ユーザーがタスクコンテナーに対して設定した制限に基づいて決まります。

クラスターに適したメモリーと CPU リソースの制限も設定できます。

2.2. コンテナーのリソース要件

タスクと Web コンテナーで、リソース要求の下限 (要求) と上限 (制限) の両方を設定できます。実行環境のコントロールプレーンは、プロジェクトの更新に使用されますが、通常はジョブのデフォルトの実行環境と同じです。

リソースの要求と制限を設定することはベストプラクティスです。なぜなら、両方が定義されているコンテナーには、より高い **Quality of Service** クラスが指定されるからです。これは、基盤となるノードにリソース制限があり、クラスターが実行中のメモリーやその他の障害を防ぐために Pod をリープする必要がある場合は、コントロールプレーン Pod がリープされる可能性が低いことを意味します。

これらの要求と制限は、Automation Controllerの コントロール Pod に適用され、制限が設定されている場合は、インスタンスの **容量** が決まります。デフォルトでは、ジョブの制御には 1 単位の容量が必要です。タスクコンテナーのメモリーと CPU の制限は、コントロールノードの容量を決定するために使用されます。計算方法の詳細は、容量アルゴリズムのリソース決定 を参照してください。

ワーカーノードにスケジュールされたジョブ も参照してください。

名前	説明	デフォルト
web_resource_requirements	Web コンテナーのリソース要件	requests: {CPU: 100m, memory: 128Mi}
task_resource_requirements	タスクコンテナーのリソース要件	requests: {CPU: 100m, memory: 128Mi}
ee_resource_requirements	EE コントロールプレーンコンテナーのリソース要件	requests: {CPU: 100m, memory: 128Mi}
redis_resource_requirements	Redis コントロールプレーンコンテナーのリソース要件	requests: {CPU:100m, memory: 128Mi}

制御ノードを個別の基盤となる Kubernetes ワーカーノードに最大限に分散するには、**topology_spread_constraints** を使用することを推奨します。要求と制限の適切なセットは、その合計がノード上の実際のリソースと等しくなる制限です。**制限** のみが設定されている場合は、リクエストが制限と等しくなるように自動的に設定されます。ただし、コントロール Pod 内のコンテナー間でリソース使用量の変動が許容されるため、**requests** をより低い量 (ノードで使用可能なリソースの 25% など) に設定できます。クラスターに 4 つの CPU と 16GB の RAM が割り当てられたワーカーノードのコンテナーのカスタマイズ例は、以下のとおりです。

```
spec:
  ...
  web_resource_requirements:
    requests:
      cpu: 250m
      memory: 1Gi
    limits:
      cpu: 1000m
      memory: 4Gi
  task_resource_requirements:
    requests:
      cpu: 250m
      memory: 1Gi
    limits:
      cpu: 2000m
      memory: 4Gi
  redis_resource_requirements
    requests:
      cpu: 250m
      memory: 1Gi
    limits:
      cpu: 1000m
      memory: 4Gi
  ee_resource_requirements:
    requests:
      cpu: 250m
      memory: 1Gi
    limits:
      cpu: 1000m
      memory: 4Gi
```

2.3. AUTOMATION CONTROLLER 設定を使用した代替方法での容量制限

Openshift のコントロールノードの容量は、メモリーと CPU の制限によって決まります。ただし、これらが設定されていない場合、容量は、ファイルシステム上の Pod によって検出されたメモリーと CPU (実際には、基盤となる Kubernetes ノードの CPU とメモリー) によって決まります。

これにより、そのノードに Automation Controller Pod 以外の Pod もある場合に、基盤となる Kubernetes Pod が過負荷になるという問題が発生する可能性があります。タスクコンテナーに直接制限を設定しない場合は、**extra_settings** を使用できます。次の設定方法は、Custom pod timeouts セクションの **Extra Settings** を参照してください。

```
SYSTEM_TASK_ABS_MEM = 3gi
SYSTEM_TASK_ABS_CPU = 750m
```

これはアプリケーション内でソフトリミットとして機能し、Automation Controller が実行しようとする作業量の制御を可能にしますが、Kubernetes 自体からの CPU スロット調整や、メモリー使用量が目的

の制限を超えた場合にリープされるリスクはありません。これらの設定は、kubernetes リソース定義のリソース要求と制限で許可されるのと同じ形式を受け入れます。

第3章 専用ノードの指定

kubernetes クラスターは、多数の仮想マシンまたはノード (通常は 2 - 20 ノード) で実行します。Pod は、これらのノードのいずれかでスケジュールできます。新しい Pod を作成またはスケジュールするときは、**topology_spread_constraints** 設定を使用して、スケジュールまたは作成時に新しい Pod を基になるノードに分散する方法を設定します。

単一ノードで Pod をスケジュールしないでください。そのノードに障害が発生すると、それらの Pod が提供するサービスも失敗します。

自動化ジョブ Pod とは異なるノードで実行するようにコントロールプレーンノードをスケジュールします。コントロールプレーン Pod がジョブ Pod とノードを共有する場合は、コントロールプレーンがリソース不足になり、アプリケーション全体のパフォーマンスが低下する可能性があります。

3.1. 特定のノードへの POD の割り当て

Operator が作成した Automation ControllerPod を、特定のサブセットのノードで実行するように制限できます。

- **node_selector** および **postgres_selector** は、指定されたすべてのキー、値、またはそのペアに一致するノードでのみ実行されるように Automation Controller Pod を制限します。
- **tolerations** と **postgres_tolerations** を使用すると、Automation Controller Pod を taint が一致するノードにスケジュールできます。詳細は、Kubernetes ドキュメントの taint と toleration 参照してください。

以下の表は、YAML の Automation Controller の仕様セクションで (または Openshift UI フォームを使用して) 設定できる設定とフィールドを示しています。

名前	説明	デフォルト
postgres_image	プルするイメージのパス	postgres
postgres_image_version	プルするイメージのバージョン	13
node_selector	AutomationController Pod の nodeSelector	""
topology_spread_constraints	AutomationController Pod の topologySpreadConstraints	""
tolerations	AutomationController Pod の toleration	""
annotations	AutomationController Pod のアノテーション	""
postgres_selector	Postgres Pod の nodeSelector	""
postgres_tolerations	Postgres Pod の toleration	""

topology_spread_constraints は、ノードセレクターに一致するコンピュートノード全体にコントロー

ルプレーン Pod を分散するのに役立ちます。たとえば、このオプションの **maxSkew** パラメーターを **100** に設定すると、使用可能なノード全体に最大限分散することを意味します。したがって、一致するコンピュートノードが 3 つと Pod が 3 つある場合、各コンピュートノードに 1 つの Pod が割り当てられます。このパラメーターは、コントロールプレーン Pod が互いにリソースをめぐって競合するのを防ぐのに役立ちます。

コントローラー Pod を特定のノードに制限するカスタム設定の例

```
spec:
  ...
  node_selector: |
    disktype: ssd
    kubernetes.io/arch: amd64
    kubernetes.io/os: linux
  topology_spread_constraints: |
    - maxSkew: 100
      topologyKey: "topology.kubernetes.io/zone"
      whenUnsatisfiable: "ScheduleAnyway"
      labelSelector:
        matchLabels:
          app.kubernetes.io/name: "<resourcename>"
  tolerations: |
    - key: "dedicated"
      operator: "Equal"
      value: "AutomationController"
      effect: "NoSchedule"
  postgres_selector: |
    disktype: ssd
    kubernetes.io/arch: amd64
    kubernetes.io/os: linux
  postgres_tolerations: |
    - key: "dedicated"
      operator: "Equal"
      value: "AutomationController"
      effect: "NoSchedule"
```

3.2. ジョブを実行するノードの指定

コンテナーグループ Pod の仕様にノードセレクターを追加して、それらが特定ノードに対してのみ実行されるようにできます。最初に、ジョブを実行するノードにラベルを追加します。

次の手順では、ラベルをノードに追加します。

手順

1. クラスター内のノードとそのラベルを一覧表示します。

```
kubectl get nodes --show-labels
```

出力は次の表のようになります。

名前	ステータス	ロール	エージ	バージョン	ラベル
worker0	Ready	<none>	1d	v1.13.0	**…,kubernetes.io/hostname=worker0**
worker1	Ready	<none>	1d	v1.13.0	**…,kubernetes.io/hostname=worker1**
worker2	Ready	<none>	1d	v1.13.0	**…,kubernetes.io/hostname=worker2**

2. ノードの 1 つを選択し、次のコマンドを使用してラベルを追加します。

```
kubectl label nodes <your-node-name> <aap_node_type>=<execution>
```

以下に例を示します。

```
kubectl label nodes <your-node-name> disktype=ssd
```

<your-node-name> は、選択したノードの名前です。

3. 選択したノードに **disktype=ssd** ラベルがあることを確認します。

```
kubectl get nodes --show-labels
```

4. 出力は次の表のようになります。

名前	ステータス	ロール	エージ	バージョン	ラベル
worker0	Ready	<none>	1d	v1.13.0	**…disktype=ssd,kubernetes.io/hostname=worker0**
worker1	Ready	<none>	1d	v1.13.0	**…,kubernetes.io/hostname=worker1**
worker2	Ready	<none>	1d	v1.13.0	**…,kubernetes.io/hostname=worker2**

worker0 ノードに **disktype=ssd** ラベルが付いていることがわかります。

5. Automation Controller UI で、コンテナーグループ内のカスタマイズされた Pod 仕様のメタデータセクションでそのラベルを指定します。

```
apiVersion: v1
kind: Pod
metadata:
  disktype: ssd
  namespace: ansible-automation-platform
spec:
  serviceAccountName: default
  automountServiceAccountToken: false
  nodeSelector:
    aap_node_type: execution
  containers:
    - image: >-
     registry.redhat.io/ansible-automation-platform-22/ee-supported-
rhel8@sha256:d134e198b179d1b21d3f067d745dd1a8e28167235c312cdc233860410ea3ec3e
      name: worker
      args:
        - ansible-runner
        - worker
        - '--private-data-dir=/runner'
      resources:
        requests:
          cpu: 250m
          memory: 100Mi
```

追加設定

extra_settings では、awx-operator を使用して多くのカスタム設定を渡すことができます。パラメーター **extra_settings** が **/etc/tower/settings.py** に追加され、**extra_volumes** パラメーターの代わりに使用できます。

名前	説明	デフォルト
extra_settings	追加設定	''

extra_settings パラメーターの設定例

```
 spec:
    extra_settings:
      - setting: MAX_PAGE_SIZE
        value: "500"

      - setting: AUTH_LDAP_BIND_DN
        value: "cn=admin,dc=example,dc=com"

      - setting: SYSTEM_TASK_ABS_MEM
        value: "500"
```

3.3. カスタム POD のタイムアウト

Automation Controller のコンテナーグループジョブは、Pod を Kubernetes API に送信する直前に **running** 状態に移行します。その後、Automation Controller は、**AWX_CONTAINER_GROUP_POD_PENDING_TIMEOUT** 秒が経過する前に Pod が **Running** 状態

になると想定します。**Running** 状態に移行できなかったジョブをキャンセルするまで Automation Controller が待機する時間を長くするには、**AWX_CONTAINER_GROUP_POD_PENDING_TIMEOUT** をより高い値に設定できます。**AWX_CONTAINER_GROUP_POD_PENDING_TIMEOUT** は、Automation Controller が Pod の作成から Pod での Ansible 作業の開始まで待機する時間です。リソース制約のために Pod をスケジュールできない場合は、時間を延長することもできます。これは、Automation Controller 仕様で **extra_settings** を使用して行うことができます。デフォルト値は 2 時間です。

これは、Kubernetes がスケジュールできるよりも多くのジョブを常に起動しており、ジョブが **AWX_CONTAINER_GROUP_POD_PENDING_TIMEOUT** よりも長い期間を **pending** の状態にある場合に使用されます。

制御容量が使用可能になるまで、ジョブは開始しません。コンテナーグループの実行容量よりも多くのジョブが起動している場合は、Kubernetes ワーカーノードのスケールアップを検討してください。

3.4. ワーカーノードでスケジュールされたジョブ

Automation Controller と Kubernetes の両方が、ジョブのスケジューリングを行います。

ジョブが起動されると、その依存関係が満たされます。つまり、プロジェクトおよびインベントリーの更新は、ジョブテンプレート、プロジェクト、およびインベントリーの設定で必要な場合に、Automation Controller によって起動します。

ジョブが Automation Controller の他のビジネスロジックによってブロックされておらず、コントロールプレーンにジョブを開始するためのコントロールプレーンがある場合、ジョブはディスパッチャーに送信されます。ジョブを制御するためのコストのデフォルト設定は 1 **capacity** です。したがって、100 capacity のコントロール Pod は、一度に最大 100 のジョブを制御できます。制御容量が負荷されたジョブは、**pending** から **waiting** に移行します。

コントロールプラン Pod のバックグラウンドプロセスであるディスパッチャーは、ワーカープロセスを開始してジョブを実行します。これは、コンテナーグループに関連付けられたサービスアカウントを使用して kubernetes API と通信し、Automation Controller のコンテナーグループで定義されている Pod 仕様を使用して Pod をプロビジョニングします。Automation Controllerのジョブステータスは **running** と表示されます。

これで Kubernetes が Pod をスケジュールするようになりました。Pod は **AWX_CONTAINER_GROUP_POD_PENDING_TIMEOUT** の間、**pending** ステータスのままになります。Pod が **ResourceQuota** によって拒否されると、ジョブは **pending** からやり直されます。名前空間でリソースクォータを設定して、名前空間内の Pod が消費できるリソースの数を制限できます。ResourceQuotas の詳細は、リソースクォータ を参照してください。

第4章 OPENSHIFT CONTAINER PLATFORM での ANSIBLE AUTOMATION CONTROLLER の設定

Kubernetes のアップグレード中は、Automation Controller が実行されている必要があります。

4.1. OPENSHIFT CONTAINER PLATFORM のアップグレード中におけるダウンタイムの最小化

アップグレード中のダウンタイムを最小限に抑えるために、Automation Controller で次の設定変更を行ってください。

前提条件

- Ansible Automation Platform 2.4
- Ansible Automation Controller 4.4
- OpenShift Container Platform
  - > 4.10.42
  - > 4.11.16
  - > 4.12.0
- Postgres の高可用性 (HA) デプロイメント
- Automation Controller Pod をスケジュールできる複数のワーカーノード

手順

1. AutomationController 仕様で、**RECEPTOR_KUBE_SUPPORT_RECONNECT** を有効にします。

```
apiVersion: automationcontroller.ansible.com/v1beta1
kind: AutomationController
metadata:
  ...
spec:
  ...
  ee_extra_env: |
    - name: RECEPTOR_KUBE_SUPPORT_RECONNECT
      value: enabled
    ```
```

2. AutomationController 仕様で、グレースフル終了機能を有効にします。

```
termination_grace_period_seconds: <time to wait for job to finish>
```

3. Web およびタスク Pod 用に **podAntiAffinity** を設定し、AutomationController 仕様でデプロイメントを分散します。

```
task_affinity:
```
```

```
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: app.kubernetes.io/name
              operator: In
              values:
              - awx-task
          topologyKey: topology.kubernetes.io/zone
        weight: 100
  web_affinity:
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: app.kubernetes.io/name
              operator: In
              values:
              - awx-web
          topologyKey: topology.kubernetes.io/zone
        weight: 100
```

4. OpenShift Container Platform で **PodDisruptionBudget** を設定します。

```
---
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: automationcontroller-job-pods
spec:
  maxUnavailable: 0
  selector:
    matchExpressions:
      - key: ansible-awx-job-id
        operator: Exists
---
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: automationcontroller-web-pods
spec:
  minAvailable: 1
  selector:
    matchExpressions:
      - key: app.kubernetes.io/name
        operator: In
        values:
          - <automationcontroller_instance_name>-web
---
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: automationcontroller-task-pods
spec:
```

```
minAvailable: 1
selector:
  matchExpressions:
    - key: app.kubernetes.io/name
      operator: In
      values:
        - <automationcontroller_instance_name>-task
```