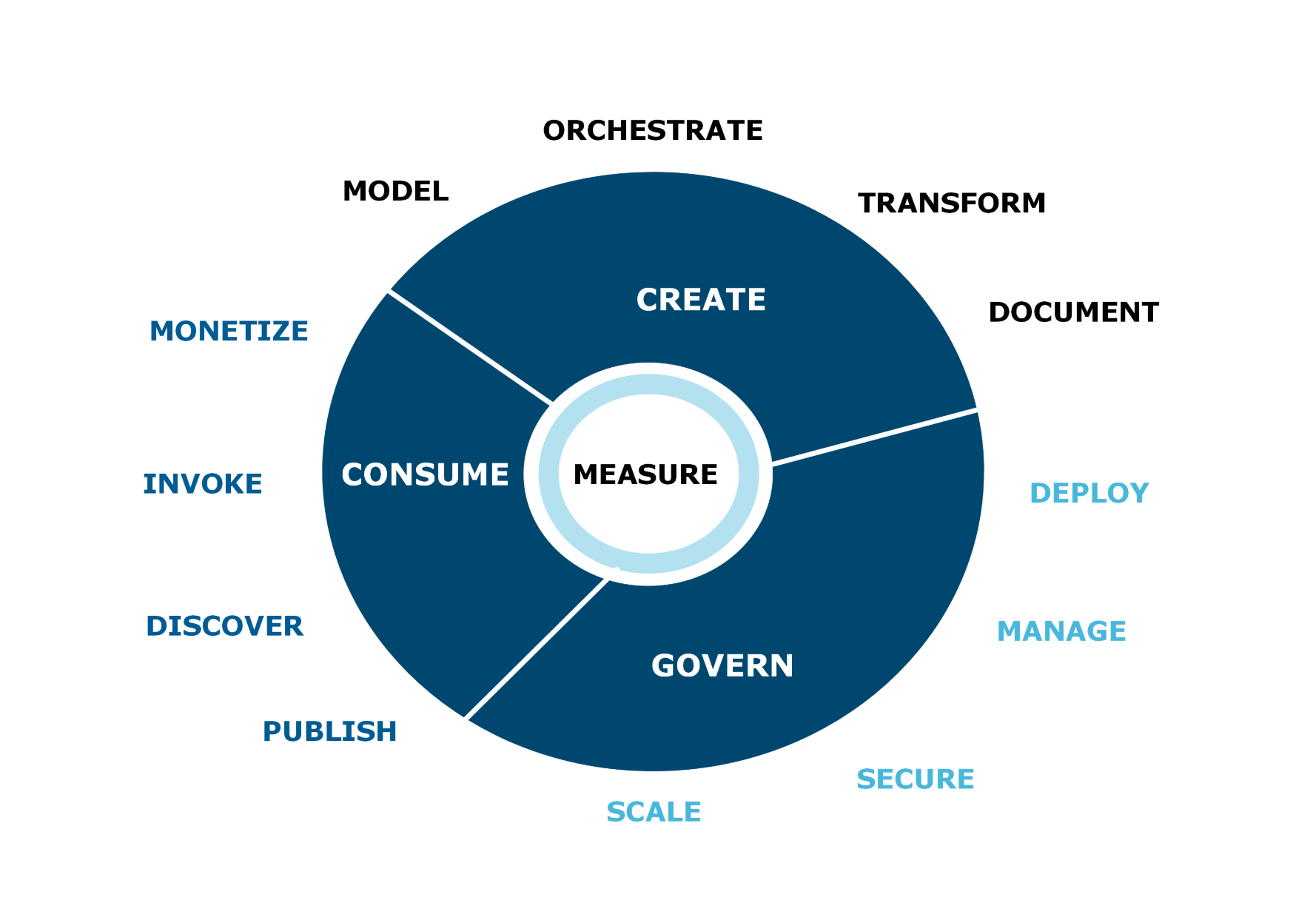
# API Deployments

API interfaces are packaged and deployed to target environments as applications, plugins, or extensions, depending on the API platform vendor. For example, in Mulesoft Anypoint Platform, an API has to be packaged as a .ZIP file and deployed to a target server or cloud instance either directly by placing the Zip file onto the server location or via Anypoint Runtime Manager (ARM). With other API platforms, an API can be deployed as a Microsoft .NET Web API application (Azure API Gateway) or a custom component written in Java.

API deployment is part of the overall API management and post-creation governance of the APIs. The diagram below shows how API deployment is the initial stage of the “Govern” stage of the API lifecycle, following the initial stage of “Create”:



No matter what format the deployment artifacts are packaged as, there are certain common API deployment requirements to be supported by the API platform:

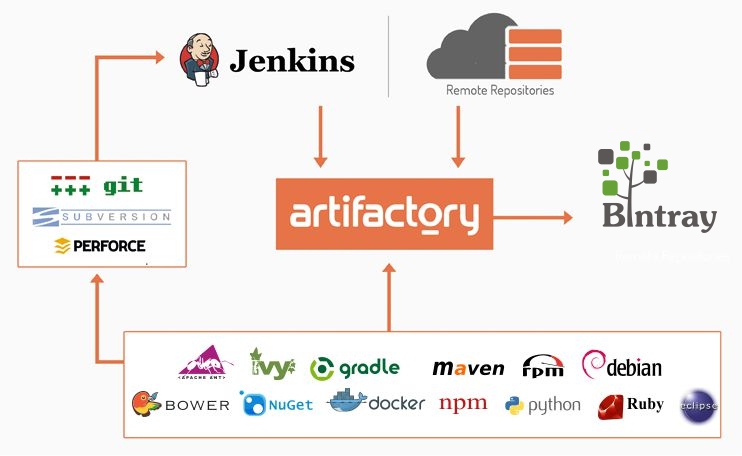
* Ensure the API deployment process is seamless and integrated into the overall API lifecycle process
* Deploy API across environments and across clusters within the same environment (for HA configurations)
* Deploy and un-deploy APIs from any environment without interrupting other deployed applications
* Deploy API versions without disrupting existing API consumers on another version
* Integrate API deployments within the overall Continuous Integration / Continuous Deployment process
* Integrate unit, regression, and system testing stages as part of the overall API deployment process
* Provide ability to rollback failed API deployments
* Integration API versioning of newly deployed APIs

Many API platforms provide capabilities to support all of the above requirements for API deployments. However, in most cases, an API governance team needs to augment API platform deployment capabilities with other deployment and testing tools to ensure the deployment lifecycle is properly managed. For example, while most API management tools provide ability to manually deploy APIs to the environments, it is recommended to utilize a robust deployment management platform such as Jenkins, Bamboo, or TeamCity. In addition, most of the API platforms don’t provide regression and system testing capabilities besides basic API console and basic built-in manually testing tools. Augmenting these API platform capabilities with a robust testing suite such as Lisa, Mercury, SmartBear, or Postman is recommended.

### Dependency Management

Dependency management becomes an integral part of the proper API deployment strategy. All but most basic APIs depend on third-party libraries and packages. During deployment of APIs, it is recommended to properly manage these dependencies on third-party libraries and packages in a dependency management platform such as JFrog Artifactory, Maven repository, Nexus dependency store, or other similar platform. In addition to managing third-party vendor-generated libraries, it is also important to manage organizational libraries, packages, and executables so that they can be shared across teams developing the APIs and microservices.

When an API is deployed to a target environment, a deployment script such as Maven or Gradle build script or a custom Jenkins script can automatically retrieve the dependencies required by the API from the dependency management solution. The diagram below shows some of the key players in the Dependency Management space, with JFrog Artifactory being the most widely used platform:



An example of using dependency management platform during API deployments can be summarized in the below steps:

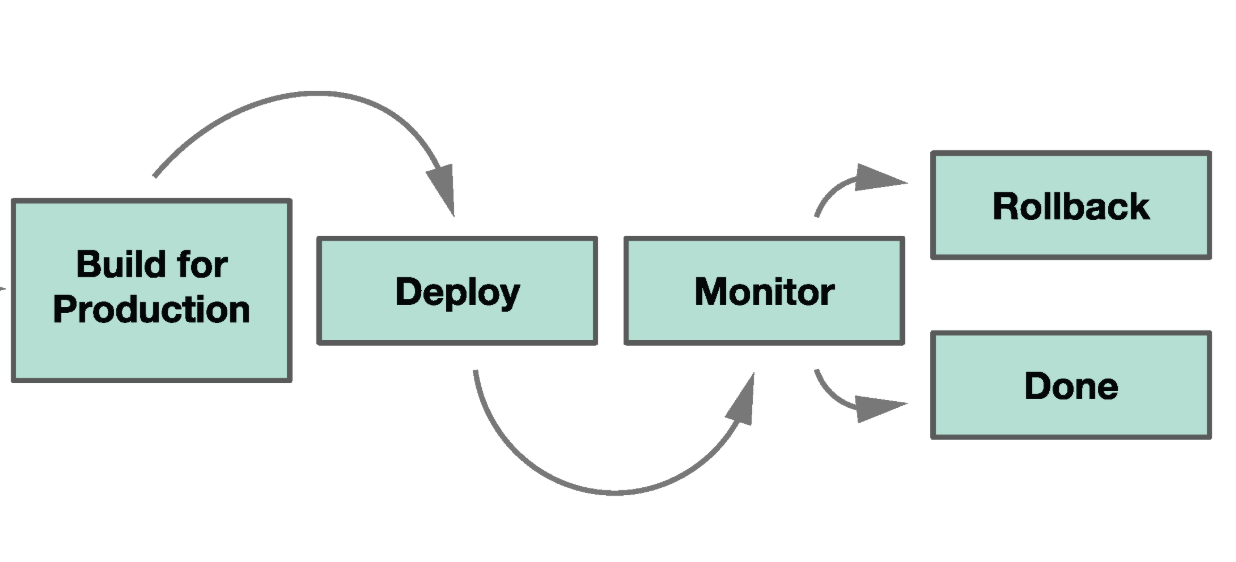
1. An API is developed using an API development environment.
2. Required dependencies for this API may include a database connector JAR file, a third-party security framework JAR, and a logging solution also packaged as a JAR.
3. These third-party dependencies are checked into JFrog Artifactory as JARs with specific versions.
4. These dependencies are then get documented inside a Maven pom.xml of the API development project.
5. Once an API project is ready to be built, packaged for deployment, and deployed, a Jenkins job is created for deploying this API.
6. A Jenkins job is triggered automatically upon developer checking the source code into a “develop” branch of the source code repository (if following the Gitflow branch management). The job retrieves the source code from the source code repository and runs the Maven command-line utility to package the API for deployment. Maven command-line utility uses pom.xml to retrieve the dependencies from JFrog Artifactory.
7. Once the dependencies are retrieved and the source code is compiled on the Jenkins build server, the API is ready for deployment to a target environment.
8. Jenkins runs the deployment either via Maven or by SSH command to move the built API deployment package to the target server.

### Deploy, Un-deploy, and Rollback Procedures

A key part of the API deployment strategy is an ability to quickly un-deploy a deployed version or rollback the deployment in case there was an error or failure on deployment. Some newer API platforms built on top of containers such as Docker provide ability to self-heal in case an API deployment fails. Most other API platforms just fail an API during botched deployment, making this API completely unavailable in the target environment.

In the case where an API platform doesn’t provide an ability to rollback a botched deployment or recover from failed deployment, the responsibility for recovery and rollback lies on the development and DevOps teams. Again, like with the actual deployment, un-deployment can be scripted via Jenkins or TeamCity jobs. A Jenkins job can be used to run a series of functional tests to validate that the API is responding to requests and provides successful responses after deployment. If, however, an API fails to return the correct responses after deployment, Jenkins job can un-deploy the latest version of an API and deploy a previous version, therefore performing a deployment rollback.

Being able to successfully un-deploy deployed APIs requires API versioning, so that specific deployed versions of an API can be easily un-deployed and previous versions can replace the failed deployments. Versioning is built into most of the API platforms, and it is a critical feature for managing multiple deployments of the same API and to support various needs of API consumers who may need different versions of the same API deployed at the same time.



### API Deployment Testing

When deployment is performed, it is important to ensure that extensive API testing is performed immediately after deployment to smoke test the deployed API in the target environment. This testing stage can be executed during the build of the API and as soon as deployment is completed. API testing should include the following testing:

* Unit testing of an API, performed during development and when an API is built for deployment
* Integration testing of an API, performed after this API gets deployed to the target environment
* Regression and UAT testing, done by a separate UAT team, also after deployment of an API

If an API deployment process is automated, unit testing and integration testing can be added as automated tasks in the overall automated deployment process. For example, if using Jenkins, a Jenkins stage in the overall pipeline can include executing unit test scripts during the initial API build stage. Then, running automated integration tests can be another stage in the pipeline right before an API is ready to be deployed. Unit test frameworks depend on the programming language or API development environment. Typical unit testing frameworks for API development include MUnit (when developing Mulesoft APIs), JUnit for Java-based APIs, NUnit for .NET, MochaJS for Node. Commonly used integration testing platforms are Postman and SmartBear SoapUI.

Most of the unit testing frameworks allow for an automated test execution via command-line or Maven and Gradle script tasks. Postman and SoapUI integration testing projects can be automatically invoked via command-line interface. Hence, an automated deployment job such as a Jenkins job can automate executions of unit and integration tests during deployments. Once an API automated testing stage is complete, a UAT team can start testing the deployed API manually from a consumer platform (i.e., a mobile device, a desktop, portal, or vendor platform that consumes this API).

A diagram below demonstrates how a Jenkins pipeline can consist of **Unit Tests** stage, **Package** stage, **Deploy** to target environment stage, and, finally, an **API Tests** stage. The first and the last stages both cover API testing for deployment, with the unit test stage running before deployment and the API integration tests stage running right after deployment is complete.

