Introduction to microServices

Continuous Deployment Platform

Dewayne Hafenstein

Principal Technical Architect

Contents

microServices

A microService is…

…any business-domain service that is implemented as a single process, accessed via an API over the network, completely self-contained and opaque, independently developed and deployed with regard to any other services (even in the same application), and usually built and deployed using automated processes.

An implementation is not a microService even if it is deployed in a Docker container unless it also meets this definition. A monolithic application in a Docker container is not a microService.

A microService is…

…used as a business-domain specific service.

…used as a single process.

…used in a independently developed and deployed manner.

…used in a self-contained manner.

…accessed using an API over a lightweight protocol (such as HTTP).

Some of the Benefits of microServices

Using microServices allows the development and delivery of features for each service quickly and independently of all other services. This generally means that new features can be developed and delivered to applications that need them, and ultimately to the customers, faster and with less costs.

Because all interaction between applications and services is defined by the API exposed by the service, the implementation of the service can vary over time and different services can use different technologies, based on what is best for their implementation.

- Services are easy to replace, extend, or alter.
- Naturally enforces a modular structure.
- Lends itself to CI/CD easily.
- Services can be implemented using different programming languages, databases, hardware and software environment, depending on what fits best.
- Good fit with Agile methodologies.
- Independently deployable, allowing multiple independent teams with different schedules.

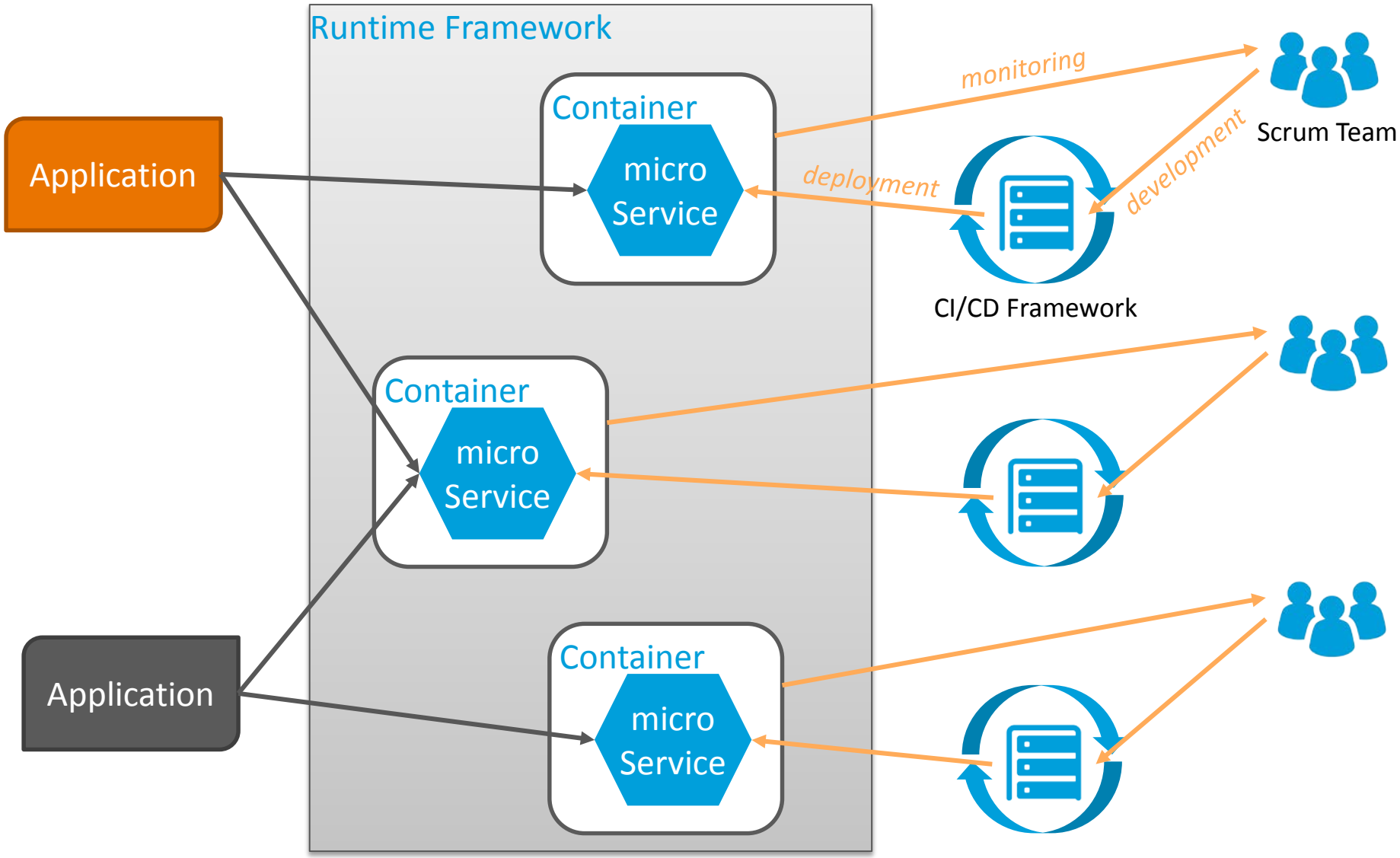
Benefits of microServices

microServices are:

- Specific domain functionality, deployed into an unshared container.
- Individually developed and supported (by their own team).
- Independently deployed (any service can be updated and redeployed at any time). Services are easy to replace, extend, or alter.

Note: All communication between the application and the microService is via a published API over a lightweight (HTTP) protocol.

microService Based Application



ms / microServices

The microService API

The API of a microService is the only way to interact with that microService.

- All microServices must expose an API.
  - The API is used to communicate with the microService.
  - The API is the ONLY way to access the service implementation and its data.

The API is a CONTRACT…

…between both the service and all users of the service.

- No other application or service can access the internal implementation of the service, OR its persistent data store, directly.
- All access to any aspect of the domain operations implemented by the service must be exclusively via the API.
- If the service changes a published behavior, it could break applications and other services using it.
- Extensions must be made such that existing behaviors are unchanged.
- The APIs must be well documented and published so all users can see what the service performs.
- API usage should be tracked and monitored to help manage the API.

Note: Generally, the microService API should be designed and use RESTful principles, unless there are reasons to use other techniques.

The microService API

microServices Communicate Using HTTP

microServices and HTTP

– microServices must communicate over a lightweight protocol.

  - microServices favor the use of “dumb” communications pipes and smart endpoints.

– Communications must be fast and simple.

  - No logic or complexity is present in the communications channel—they are generally just movers of data, and the “smarts” for how to locate and call services is in the service itself.
  - HTTP protocol is sufficient in most cases. When guaranteed delivery is required, other communications protocols are used (such as Kafka)
  - HTTP is stateless and point-to-point.
  - HTTP is fast and is well understood.

Note: HTTP does not guarantee delivery of a packet. That can be handled in the service by retry logic, or by the use of a guaranteed delivery communications layer. The connection between an application and the service, or between a service and another service, should be a fast, lightweight, and simple protocol, like HTTP.

– Communication allows service portability—the services can be moved around as needed without causing an outage.

  - This allows the service to be automatically scaled up or down, and instantiated whenever there is a system failure.
  - No applications or microServices are aware of the locations of any other microServices except by looking them up in a registry when they need them. There is no location dependency, and thus the locations can change and the network allows them to continue to function as normal.

microServices and HTTP

microServices are Opaque

ENCAPSULATION

- microServices must be encapsulated—this is also known as “opaqueness” or being a “black-box.”
  - All interaction with the microService must be via it’s API.
  - Encapsulation simply means that all of the logic and data that is needed to implement the service is not exposed to any outside “user” of that service.
    Note: It DOES NOT mean that all of the implementation has to be performed inside a single process or image.
  - Encapsulation allows the implementation to change without affecting the interface (API) and any users of the service. This allows:
    - New features to be added at any time.
    - Database schemas to be adjusted, corrected, or extended without affecting users of the service.
    - Corrections and feature removal to be done.
    - Refactoring for performance or supportability.
- Encapsulation in this context is functionality.
  - All implementation AND data of the service must be accessed only by the API.
  - The service may use external facilities, such as data bases, but their content must only be accessed by the service. Any application, or other service, that needs the data must access it via the API.

Encapsulation

microServices are Opaque

DATA STORES

- It is highly likely that a service needs to access persistent data stores.
- These databases would still exist as they do today in a non-microServices implementation.
- The essential difference is that the data stores for the services would be completely owned and accessed only by the service.
- No other service or application component would access these databases directly. All access must be through the services API.
- This also means that services should not “share” databases.
- Doing so will tend to invite coupling between the services at the database level.

Data Stores

microServices are Deployed to Containers

microServices run in their own containers.

- microServices utilize the benefits of service virtualization via containers.
- A microService deployed into a container has all of its dependencies provided for it by the container.
- The infrastructure is managed by the container, so the service focuses on the business processes and need not be involved in it's management.
- Also, the service is deployed into its own, unshared container. There are no dependency conflicts that need to be resolved. In addition, any configuration needed by the service is also unshared, so conflicting configuration settings are not an issue.
- The separate container per microService approach allows complete isolation.
    - Each service's lifecycle is unique and can be managed separately.
    - Each service has its own dependencies and configuration managed by its container.
    - One service cannot inadvertently interfere with another service.
    - Deployment is simplified because conflicting dependencies and configurations are not a problem.
- Containers manage the infrastructure, making the service portable.
    - The services are not aware of, or involved in, any infrastructure management.
    - The containers can be run on any supporting platform.
    - The service is simplified because all infrastructure management is removed.

microService Containers

microServices are Elastic

microServices are “discovered.”

- The location of a microService is not fixed.
- microServices are elastic by nature—they are started when needed, and can be stopped when no longer needed. This elastic behavior allows microServices to adapt to workload, failures, and other dynamic events as they happen.
- The runtime framework could create multiple copies based on demand.
- In the event of a failure, the runtime framework could start a new instance on a different server, or possibly in a different data center. The runtime framework may start and stop instances because of…

  …outages

  …failures

  …scalability
- Applications must lookup the desired services. microServices register their existence with a central repository, which application and other services can access to lookup services. The registry returns the location of the service, which can then be invoked to perform the operation(s) desired.

microServices are Observable

microServices must be managed.

- The very nature of microServices makes them hard to manage.
- They can be instantiated on any platform, any where, at any time based on lots of factors. This dynamic nature of the microService makes keeping track of logs, performance metrics, and the very existence of the microService itself a challenge. The logs may need to be accessed to diagnose problems or observe behavior.
- Performance metrics may need to be accessed to verify performance.
- It may be necessary to access the microService to debug a problem.

Since microServices are elastic and can move, finding them can be an issue.

- The runtime framework comes to the rescue! It captures all microService logs and performance data and sends them to a central location for access.
- The framework knows where the microServices are running at all times, and it can keep track of that information for us.
- Also, if we need to know where a specific instance of a microService is running, the registry can tell us that. The runtime framework keeps track of logs, even if the microService moves.
- The runtime framework gathers and correlates performance metrics.
- The service registry provides information about where services are running “right now.”

microServices Are Independently Developed

microServices are developed and supported by separate teams.

- One microService is developed and supported by one team.
- A team could support multiple microServices if they are small enough and the team has the knowledge and capacity.

Since microServices are all independent of one another, and the only access to them is via their API, the development and delivery of the microServices need not be synchronized. If one application needs a new feature, that feature can be added and the new version deployed without impacting any other users of that service.

This is true only if the API maintains its behavior and does not change for existing users of the service. New functionality is implemented by adding new operations to the API, or adding new options to existing behaviors, or by adding new optional data items to existing data structures.

Existing behavior must be maintained, but if that is done, the ability to deliver new functionality independently and as fast as possible speeds delivery of functionality to customers.

microServices have their own delivery cycles.

- Each microService is released on its own schedule.
- Dependent upon the user stories being developed by that team and their priorities and availability.
- microService releases do not have to be synchronized; they happen when they need to happen.

CI/CD

microServices are constructed and deployed using automation

To speed delivery the construction, testing, packaging, certification, and releasing processes need to be automated. The automation takes place in both the integration and verification of changes to the code base, and in the construction, packaging, release, and deployment of the services. This is collectively referred to as continuous integration and continuous deployment.

It is important that the processes used to develop and deploy microServices be automated.

- Much shorter time to deploy new features
- Repeatability and automation eliminates mistakes

Continuous Integration reduces integration issues

- Multiple developers making changes to a service code base can create conflicts.
- The chances of conflicts between changes increases as the time between integration increases.
- CI seeks to integrate changes, build the codebase, and run all verification tests as fast as possible to give feedback to any conflicts or problems quickly and prevent the conflicts from escalating out of control.

Continuous Deployment delivers new functionality quickly

- Produce software in short cycles.
- Software can be reliably released at any time.
- Helps reduce the cost, time, and risk of delivering changes by allowing for more incremental updates

What is a microService?

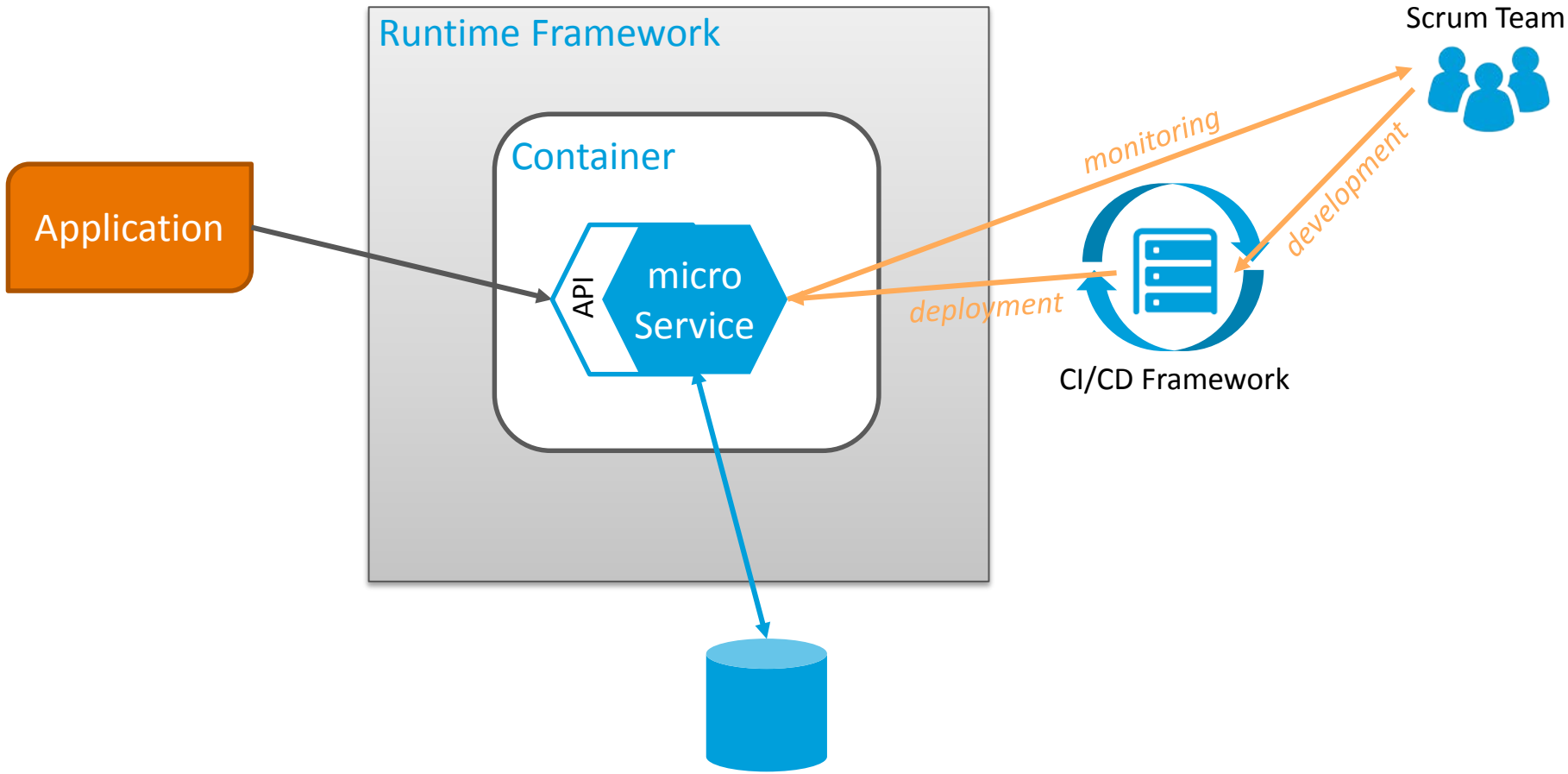
So to recap, what makes a service a microService?

- A microService is a domain-specific business service (implements domain business functionality) that is independently developed and supported (from any other service and by its own team), constructed and deployed using automation, deployed into its own, unshared container, is encapsulated (no other application or service can access its state or behaviors other than via its API), and is accessed by its API over a lightweight protocol.

So, a possible definition of a microService might be:

- A microService is an architectural style; a way to structure program functionality that allows for rapid development, deployment, and enhancement, improving quality and time to market.
- Compatible with multiple languages, microServices are based on modeling the business domain as a set of reusable business functions, or "services“, communicating over lightweight protocols and exposing APIs to access them.
- The services are defined based on the types of activities performed in the business, with each providing different functionality to support the business.
- Applications are constructed by reusing these services. Essential processes involved include CI/CD automation and use of container-based code deployments.

What is a microService?



Contents

What is an Application?

An application implements a specific business process or function.

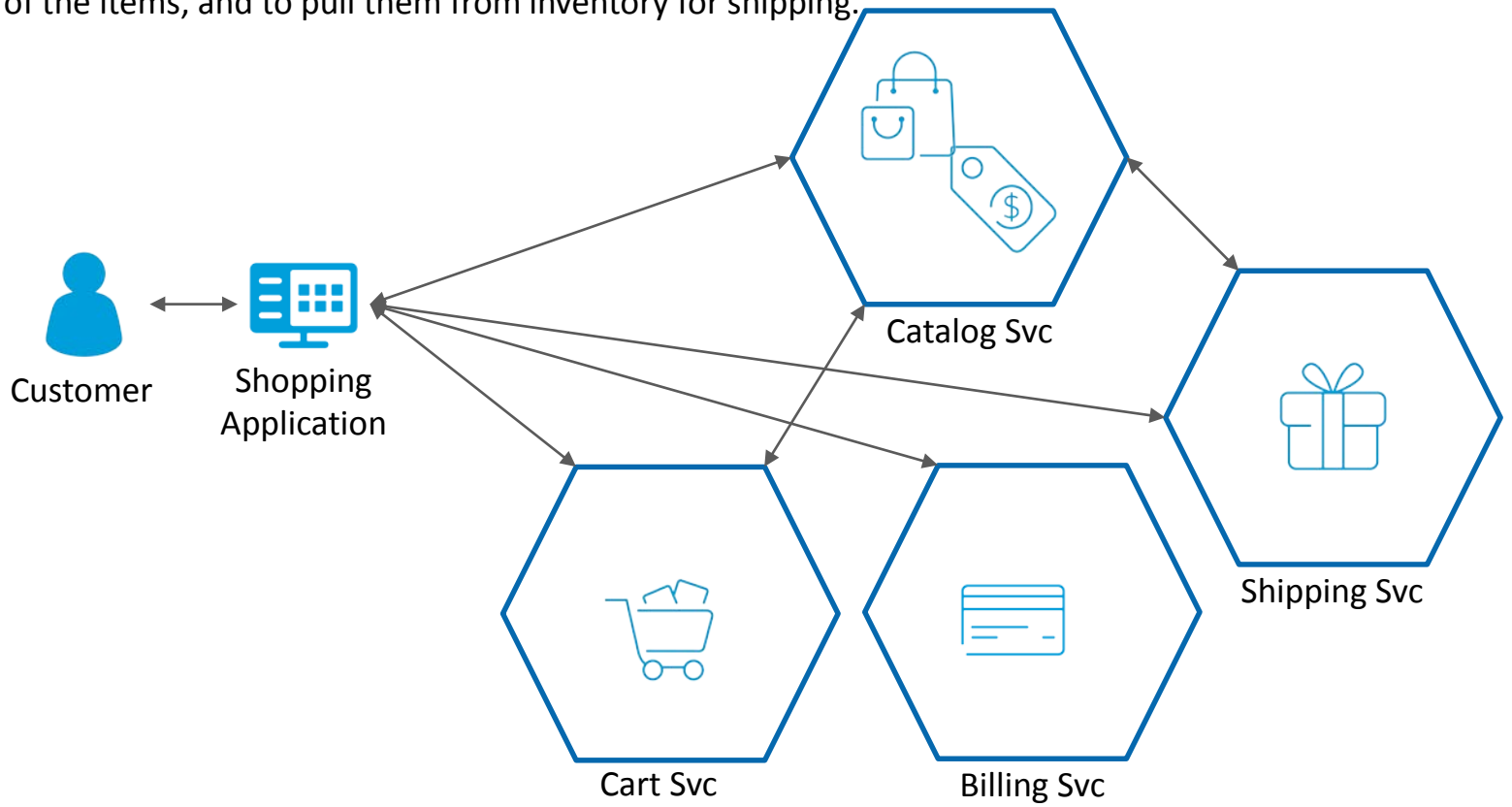
- Applications can be implemented using any architectural style, but this course is about microServices, so the focus is using microServices in an application.
  - Applications are usually “customer facing.”
  - Applications coordinate the operations of the services to perform some business process.
- The choice of style depends on a lot of factors. However, if the choice is to use microServices, then the application delegates certain behaviors to various microServices to provide that processing. In this case, the application coordinates the use of the various microServices to achieve its goals.

Specific Business Process or Function

What is an Application, continued

Application Example

- The business domain may have many processes, each using different combinations of services to perform those processes.
  - For example, an online shopping application may interact with the catalog service to obtain lists of items for display, the billing service to actually charge the customer for the items, and a shipping service to ship the contents to the customer. A cart service may manage the shopping carts, providing long-term retention and eventual destruction in the event the user does not complete the transaction.
- Additionally, services may interact with other services. For example, the shipping service could interoperate with the catalog service to obtain the weight and size of the items, and to pull them from inventory for shipping.



Specific Business Process or Function

Service Reuse

Services represent specific domain functionality that are used by applications to implement business processes.

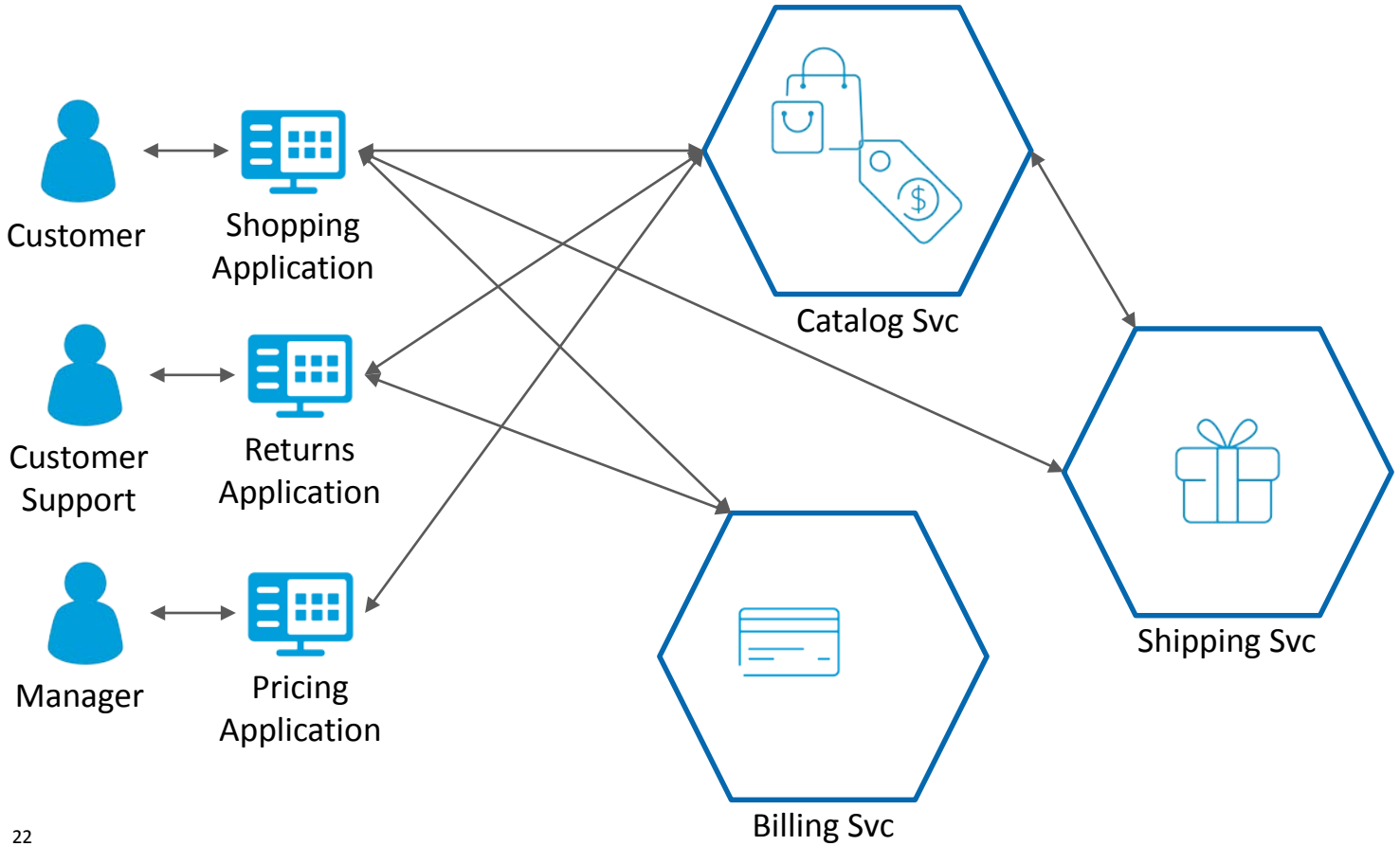
- These additional applications would provide different processes to support the business, but would reuse the same services to implement those processes. For example, a customer returns an order that they previously purchased. In this case, the customer support technician would return the items to inventory and make them available for sale in the catalog, and process a refund using the billing service. A manager may need to adjust the prices of items for sale, view the current inventory, or start and end sales.
- These are all very high level examples of how applications would be constructed using these services.
- The actual business processes for a real retail store operation would be far more involved, but the essential concepts are the same.
- It is also important to note here that the use of these services and the definition of the services themselves is a consequence of domain-driven design and the choice of the domain model.
- This should help to illustrate why it is so important that the domain model be integrated into the design and development of the services and the applications, and that it live for the life of the services. The applications, services, and domain model are interrelated and cannot (or should not) be separated.
- This means that the same service could be used by many applications, or by other services as well.

Service Reuse

Service Reuse

Services represent specific domain functionality that are used by applications to implement business processes.

– Extending our online shopping application example, the business may also need applications to enable processing returns and managing the price of items, especially sales and markdowns or markups over time.



22

Creating a New Service

Services are driven by the domain model.

- The definition of the services, what they do, the data they operate upon, and the services they perform are all dictated by the domain model, not by the application. The domain model was developed to represent the business domain, its data, and the fundamental capabilities that must be accomplished within that domain. Applications should be able to reuse these services if the domain model is correct. If you find that you need to develop an application that does not fit the domain, then either the domain is wrong or the application, and either or both need to be revisited.
- To use services correctly, the application developers need to understand the domain model that the services are part of, and that their applications are also part of. If the application cant use the services, or the application does not fit in the domain, then the design is incorrect or the application is in the wrong domain, or a combination of both.
- The domain model determines what services are needed and what they do.
- Everyone needs to be aware of the domain model.

Creating a New Application

Applications use microServices to perform a business process.

– If an application needs a service, there needs to be a way to “find” information about the service.

- The existence of microServices must be published.
- The details of their APIs must be published.
- The domain model that they are part of needs to be available.

The essential qualities of the services are they vary less frequently than the processes do. For example, an online sales business (like Amazon for example) would likely always need the ability to manage a catalog, and to perform billing and shipping functions. These services would not vary much over time, but the ways they are combined and used could.

If you find that you are always adjusting service implementations or adding one-off use cases for specific applications, that should be a red flag that says the service is not defined correctly. Services should not vary much. If anything, new capabilities should not be minor variances of existing capabilities. Doing so is a recipe for disaster.

Developing a new application, or adding a capability to an existing application, where a service is needed should reuse an existing service. This means that services and their capabilities needs to be published and maintained in a way that everyone is aware of the service and knows how to use it. The existence of the service, its API, and its domain model need to be available to all application developers.

Creating a New Application

New applications can be developed at any time.

- They can be specialized to specific customers, or to enhanced or changed business processes.
- Business processes can change or be created over time for many reasons, including regulatory, technology changes, or competitive pressures.
- Processes also may die for the same reasons. These business processes are all parts of the same business domain and as such, are realized by using the services of the domain in different ways. microServices should be reusable.
- microServices should vary much less frequently than the applications.
- You should not need to develop new microServices for new applications very often.
- You can extend or refactor microServices easily without impacting the use of that microService.

Applications from microSevices

Contents

Domain-Driven Design

Domain-Driven Design (DDD) seeks to solve several problems.

DDD primarily seeks to establish a common understanding and representation of the business domain, and simplification of the problem at hand to the simplest model that is accurate and complete and can be used to describe the software product. This is done to facilitate common understanding of the problem and the solution so that subject matter experts, analysts, architects, developers, testers, and support engineers can all have a common understanding, interact with each other, contribute to the product, and validate that the design is correct.

- The converse, as was attempted in the past, was to create a globally consistent model across the entire business, resulting in a brittle model that was easily broken by differing contexts within the domain.
- DDD recognizes that there are different contexts, or perspectives, in the business and drives to identify them, isolate them, and develop a ubiquitous language and model for each context. There is no single, large, global model.

Domain-Driven Design, continued

Domain-Driven Design (DDD) seeks to solve several problems.

One of the challenges in any business domain is that there are often multiple perspectives which sometimes have competing needs or desires. These different perspectives are similar to the different faces of a Rubik's cube puzzle. The domain is still a single thing, but there are multiple ways of looking at it. These different perspectives are known as “contexts”, and the Domain-Driven Design includes some guidelines on how to manage these different perspectives using a concept called a “bounded domain model.”

- Allows people with different interests and skills to work together on the analysis and design.
  - It is impractical to have everyone be an SME, analyst, architect, and developer.
  - A common vocabulary allows SMEs, analysts, and developers to understand each other. It also documents the design and the implementation using the same terminology.
- Creation of a simplified domain model…

  …facilitates understanding.

  …serves as the central organization of the design.

  …allows all team members and SMEs to understand and verify the design by creating small domains of consistent definitions.

  …manages multiple, possibly overlapping or competing, perspectives.

Essential Concepts About DDD

The Domain Model and the design shape each other.

- In domain-driven design, the SMEs and analysts develop a common vocabulary of terms and understanding as they work through the problem domain. The terms must be the same terms that are already familiar within the domain; do not create new terms, that only causes confusion.
- As the SMEs and designers develop an understanding of the essential behavior of the domain, they start to develop the domain model.
- As the model is being developed, a preliminary design will start to take shape. The model and design are tightly coupled to each other, and each are refined as the other changes. The implementation also uses the same vocabulary as the model and helps to tightly couple the domain model into the implementation. The model and design are incrementally developed together and drive each other.
- The model helps provide understanding during maintenance and ongoing development.

The model is used by all team members and SMEs.

- Every member of the team uses the model to describe and understand the domain, and the design and implementation of the product. The model allows everyone to communicate with a common understanding.

The model is distilled domain knowledge.

- The model captures and represents the way the SMEs, analysts, and developers choose to think about the domain.
- It captures the terms, concepts, and relationships in the domain.
- The domain model will always consist of a data model that describes the entities, aggregates, and value objects of the domain, and may also use many other diagramming techniques to describe processes, user interactions, state changes, components, and so forth. The exact design artifacts will vary greatly from one domain to another and are developed as needed.
- The model is not created once and discarded, but rather becomes a living source artifact of the product. It is essential that the model be maintained for the life of the product, and any ongoing development and maintenance apply to the model as well as the implementation.

A common vocabulary is developed and used throughout.

- The terminology and understanding of the domain model must be expressed in terms of the domain, and become a shared vocabulary used all the way through the product.

Contents

Agile and microServices align well.

Using agile methodologies to develop microServices is relatively well aligned. Agile methodologies and microServices have complementary goals. Agile is an attempt to develop software faster and more iteratively, and microServices are a good means to implement that goal. However, there are some things to watch out for…

Agile	microServices
Agile teams need to be small and focused.	microServices need to focus on domain services.
Agile is intended to make it faster to develop software.	microServices enables faster delivery of features because of the independent, encapsulated structure of microServices.
Agile builds only what is needed when it is needed.	microServices enable the delivery of these incremental capabilities faster.
Agile focuses on quality through test-driven development.	Continuous integration helps implement that. Continuous integration is a major factor in microServices.

Design Pitfalls

Design Models

While most people associate agile with having zero artifacts, the realities of business are that there are probably a few items that must be produced including training materials, user guides, etc. But most certainly, especially with microServices, the domain model and design must not only be produced, but maintained and used for the entire life of the services. Design is one of the aspects of using agile methodologies with microServices that needs to be managed.

- **Agile methodologies** (with a few exceptions) do not recommend doing NO design, but rather doing only the design and creating only the documentation that is necessary.
- **Agile methodologies** tend to de-emphasis formal design, while microServices seems to require good domain understanding and identification of domain services.
  - Agile attempts to eliminate ***unnecessary*** design, and to favor light-weight, incremental development and frequent delivery.
  - When developing microServices using agile methodologies, good domain-driven design is necessary!

With microServices, the accurate modeling of the domain and choice of services is critical to reuse and the elimination of features that span services. Poor modeling of the domain and selection of the service boundaries may cause features to be partially implemented in more than one service. All of the service teams need to be invested in, aware of, and contribute to the domain model, so some means of managing this shared knowledge and evolving it over time is needed.

- **Domain-Driven Design** requires that the domain model be a living artifact, but services are only part of the domain model.
  - There are potentially several solutions to this situation.
  - The domain design can be shared across services and scrum teams. *This means that all of the teams contribute to and understand the model.*
  - The domain design can be managed by a separate team and features “given” to the various scrum teams for implementation.

Implementation Pitfalls

Loosing Sight of the Forest

One of the things that needs to be carefully managed is the fact that microServices are usually implemented by different teams, and that these microServices are part of a larger business domain. This causes the teams to become myopic in their microServices and they can loose the context of the overall business domain. This can be offset by ensuring that all teams working on all microServices are fully aware of the entire business domain model, have access to all of it, and are kept up to date on new features that may impact them.

– It is easy for developers in different scrum teams, working on different microServices, to loose focus on the “big picture.”
  - This is especially true because a microService is not the entire application, and a new feature being added to a microService is likely only part of the complete implementation.

Technical Debt

Another implementation issue that can arise is the development of technical “debt”. This is a term that was coined to represent all the little hacks and workarounds that get injected into code to get it to work “good enough” to be released, but which will need to be refactored or completely redone later. This has a tendency to accumulate over time to the point that in order to create a new feature, a significant effort may be required to correct these “hacks” or partial implementations. This debt does not contribute to the overall feature set or quality of the product, and becomes a liability. Accumulation of technical debt tends to increase as speed is increased. This should be managed closely.

– Debt is created when the implementation takes shortcuts, or leaves in partial implementations, usually in order to meet deadlines, such as hard-wired or naïve implementations, algorithms, or work-arounds.
  - Since applications are constructed from multiple microServices, technical debt can cause unforeseen issues.

Technical Debt

Support Pitfalls

Refactoring across microServices is harder.

Refactoring “across” services occurs if the business activity requires changes to two or more microServices in order to realize the feature. In these cases, coordination of these changes can be more difficult. It needs to be remembered that every microServices could be reused in multiple, different applications.

Changing an API behavior may have unforeseen consequences for other applications. This means that any refactoring that may be needed to correct API behaviors or to align the microServices may be much more difficult when it involves multiple microServices. This is another reason why domain analysis is critical.

- Avoid features that are implemented across services.
- Services should be distinct, self-contained, and independent.

Refactoring across microServices

Contents

Development and Delivery

Developing applications using microServices requires attention to several issues.

In order to be successful using microServices, you must understand and manage several key concepts and approaches.

Domain-Driven Design (DDD)

DDD is essential to identify *what* microServices are needed. The design must be kept current and maintained as part of the lifecycle of every microService used to implement the domain.

There is a tradeoff between the size of the microService and the ability to independently design, develop, and deliver the microService. If a domain is broken up into multiple microServices (the usual case), then the design needs to transcend the boundaries of all of the teams and live in all of them in some way. DDD involves the following concepts:

- Analysis of the domain
- Assignment of microServices to construction teams
- Ownership and management of the design

Methodologies

microServices are a natural evolution from agile methodologies, making the development and support of microServices easier and more able to realize the benefits of microServices when using agile methodologies.

- Works better with Agile
- Works better with DevOps—which is also an evolution from agile methodologies and is often used with microServices approaches.

Development and Delivery, continued

Developing applications using microServices requires attention to several issues.

Processes and Controls

The processes for developing and deploying microServices should be automated as much as possible to make the process repeatable and rapid, and to facilitate the independent and short delivery of functionality to the users.

The use of automation does not eliminate the need for controls, audit, and approvals to proceed. Therefore, the process used in CI/CD needs to be monitored, controlled, and allow for audit and external synchronization gates.

- CI/CD pipeline
- Gates and Auditing

Automation

These gates allow for management approvals, reviews, or any synchronization with an external process. Additionally, once microServices are deployed, their use, performance, and logging of activity needs to be managed, collected, and made available for analysis.

- CI/CD
- Monitoring

Infrastructure and tooling needed

microServices needs some infrastructure and tooling.

To support the processes needed to be successful using microServices, some infrastructure and standard tooling needs to be in place. This standard infrastructure provides the environment and tooling that are used by all members of the microService design, development, operations, and support team.

Continuous Integration

The management of the code base and the continuous integration of enhancements and changes requires a SCM repository and tooling to support it. This is used by the developers to create, modify, and support the code base. It is also used by the build automation to pull the code to be built. It is also used by quality analysis tools and other verification tooling that may be used. The SCM repository becomes the central point of control for all source materials, which would include not only the source code, but may also include configuration information, design artifacts documentation, and any other "source" like materials.

- SCM repository with the ability to managed collisions and merging
- Automated builds
- Automated testing and quality checks

Continuous Deployment

- Release management
- Controls and audits
- Repository Management
- Configuration Management

microServices Infrastructure and Tooling

Infrastructure and tooling needed, continued

microServices needs some infrastructure and tooling.

Runtime Support

The runtime environment for the services needs to be able to manage the containers, create new instances when needed, and remove instances when no longer needed. The runtime management needs to be able to manage failures of the environment and automatically create new instances of the services to continue operation. This also means that services must be registered into a runtime registry that can be queried to locate services when needed.

- Environments

Visibility

Since services move around for various reasons (scalability, outages, etc.), the ability to diagnose issues in the services is more complex. This means that some capability must exist to capture logs and metrics from services where they are running, when they are running, and accumulate these in a centrally accessible place. This allows an engineer to access the log feature and analyze log output from a service no matter where it is running.

- Log availability
- Performance monitoring
- Analysis and Debugging

Infrastructure and tooling needed, continued

microServices needs some infrastructure and tooling.

Standards

- Containers
- Frameworks

Ease of Use and Bootstrapping

Setting up microServices can be an involved process. Creating the microService project is just the tip of the iceberg. There are SCM repositories, CI/CD processes, runtime, and all sorts of other management infrastructure that may need to be configured. This could include security, product management and accounting systems (like MOTS), and others. Using a template-based approach makes the initial set up easier, since all the definition of the processes and configuration can be included in the template.

Templates

- Examples
- Documentation

CDP

CDP is the AT&T standard infrastructure, tooling, and processes for microServices.

- CDP is the standard set of tools, frameworks, and processes for developing, deploying, and monitoring or supporting microServices at AT&T.
- This system consists of many parts, including a template-based initialization capability, CI/CD process support, testing and verification tools, SCM repositories, catalogs, runtime management, log management, and metrics management.
- CDP is a complete framework used to manage all aspects of the microService development, deployment, and support.

Bootstrapping

Creating a microService project initially can be challenging.

CDP provides a template-based approach to getting started.

– Use templates to create and initialize a project.
  - Creates the project structure.
  - Registers the project with the SCM repository (Git).
  - Creates the needed jobs in Jenkins.
  - Links Jenkins and the SCM repo.
  - Creates the pipeline process used to control the CI/CD cycle.
  - Sets up security.

CDP Template Approach

Software Configuration Management

Git is the repository for SCM used by CDP.

- CDP uses Git.
- The developers use Git to manage their project artifacts.
- The CI/CD pipeline uses Git to build and manage the service project.

Git Repository

Security

CDP uses AAF to manage security.

- An AAF namespace is created for each service project.
- The AAF namespace is created by the template used to create the project.

AAF Security Management

CI/CD Pipeline

The CI/CD process is called a “pipeline”.

The processes performed by the CI/CD framework for a specific microService are called a “pipeline.” The pipeline defines how the microService is constructed, verified, packaged, and deployed.

The pipeline may include manual synchronization points, called “gates”. These manual synchronization points stop the pipeline process until the gate is satisfied. A gate could be used for a management approval, or for synchronization with external processes. The exact pipeline will depend on the management processes and requirements for the specific service, and means that different services may have different pipelines.

- It defines the process performed.
- It defines the “gates” in the process.
  - A gate can be a management approval, external synchronization point, or any place in the process where it needs some external operation before it can proceed.
  - The gates can be different for different types of projects.
- The pipeline is initially created by the template used to create the project.

Continuous Integration

Jenkins

- The tool used to perform continuous integration
- Jenkins executes the CI/CD pipeline process.
- As code is changed, the following occurs:
  - Project is built.
  - Automated tests are run.
  - Code metrics and other quality analysis tools may be run.
- Results are provided back to developers.

Continuous Integration

Code verification is performed automatically.

Code verification is any process or sequence of processes that is used to verify that the code is acceptable and ready to move on to the next step (deployment). Verification can take any form that is appropriate for the specific technologies being used, and in many cases may employ combinations of verification tools.

Verification can consist of unit testing tools, quality analysis tools, complexity checkers, standard verifies, review tools, and others. For a technology such as Java, this might include Junit, Coburtura, CheckStyle, SONAR, or many other tools. For other technologies, different tools and facilities will be used.

- Code verification is performed each time the project is built during the CI processes:
  - Automated testing is performed.
  - Automated code metrics are performed (if any).
- The results of code verification are quickly made available to the developer(s).
  - Allows developers to correct integration mistakes.
  - Keeps size of integration effort low.
  - Helps ensure implementation quality and correctness.

Continuous Deployment

The verified codebase needs to be available for rapid deployment.

- One of the goals of microServices is rapid delivery of functionality and independent delivery of services.
- Verified code should be able to be deployed at any time.

CDP uses Docker Images.

- The code base is built into a Docker image.
- The image is published into an image repository.
- The image can be installed at any time into the runtime framework.

Docker Containers

CDP will run each service in its own Docker container.

- The containers provide isolation.
  - Each service is separate from every other service
  - All dependencies for a service are unique to that service

- Docker containers can be portable, allowing the service to run anywhere.

Docker Containers

Runtime Framework

CDP uses Kubernetes to manage Docker containers.

- Kubernetes clusters will run the Docker containers for each service.
- Kubernetes will inject the configuration into the container at runtime.
  - Configuration settings are maintained outside of the container or image, and are provided to the container when it is started on a Kubernetes cluster by Kubernetes.
- Kubernetes .inject “secrets” into the container at runtime.
  - Secrets are items such as user ids, passwords, keys, or any other values that you want to keep separate from the image and need to be set at runtime.
  - Managed similar to configuration settings, but are secured and protected by Kubernetes.

Visibility

Containers register their service's existence.

- Containers register the service when started.
- Containers unregister the service when terminated.

The registry can be queried.

- A developer or support engineer can query the registry to see where services are located, how many, etc..

Logs and metrics are captured and exposed in a central location.

- CDP captures all logs from all services and record those logs in a central, easily accessed location.
  - Allows engineers to view logs from any service regardless of where it is running.
  - Allows engineers to view and analyze performance metrics from services regardless of where they are running.
  - Logs and metrics are accumulated in near real-time.

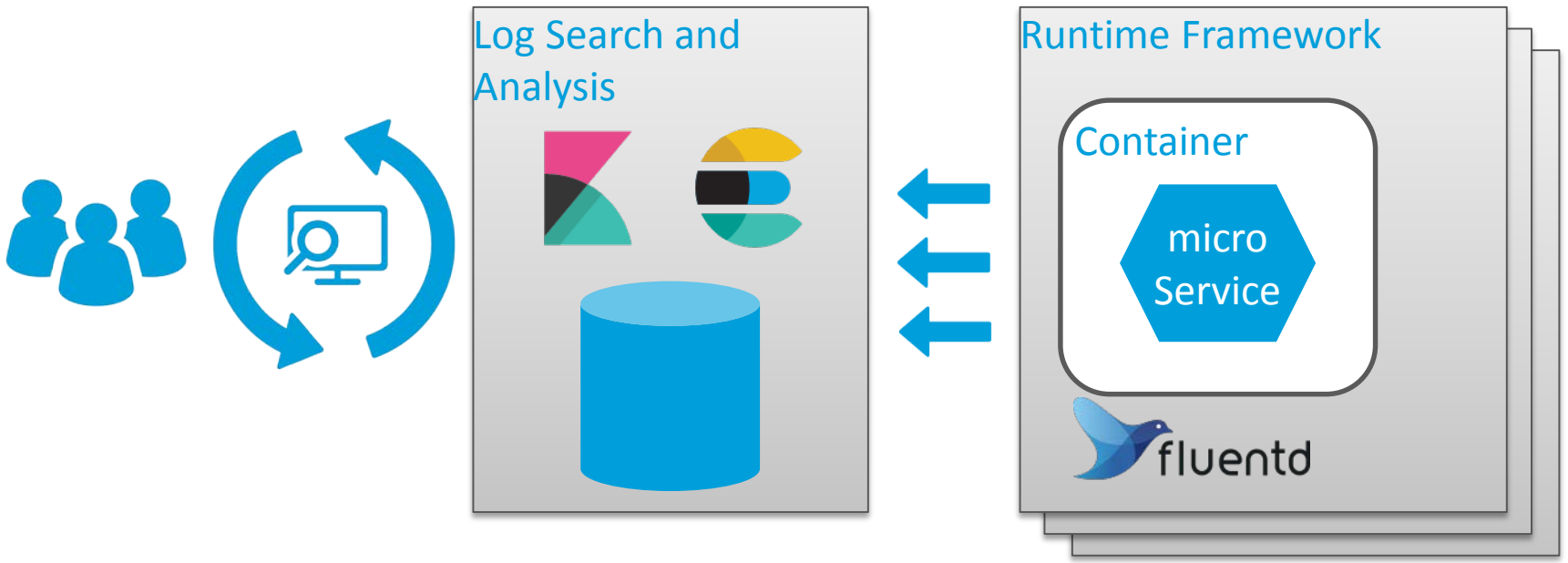
Log Management

CDP captures and manages container logs.

- The microService is deployed into a container that has CDP runtime components automatically installed.
- These components include Fluentd, which ***captures*** all log output from the microService through the container and ***forwards*** this log data in real, or near real time to an analysis server (or cluster of servers). The data is stored in a central database and can be analyzed using Elasticsearch and Kibana.
- Engineers can locate logs using advanced search capabilities and analysis tools to view, search, and interpret the logs.

Since the microServices are running in containers which may be created, destroyed, and moved based on the operational environment (scalability needs, failure recovery, outage, etc.), the logs cannot be accessed inside the containers themselves.

It is also very difficult for a user to know which container will be used from one request to another. So, there has to be a way to perform more sophisticated log analysis in a central place.



Log Management

Metrics Management

Metrics record information about resource utilization, performance, load, and alerts.

Metrics is similar to log management. For the same reasons as logging, the runtime metrics have to be captured and forwarded to an analysis system. The CDP runtime framework will have Prometheus already installed which will gather the metrics and forward them to a central Prometheus server for analysis.

The user can then use Prometheus and Grafana to search, analyze, and interpret the metrics, regardless of where the service and container may be running. Additionally, Prometheus integrates with monitoring and alerting, such as Nagios to generate alerts and alarms whenever the services are not operating correctly.