# Best Practices for Application Development

## Developing Applications with Google Cloud Platform

CLOUD STORAGE, CLOUD PUB/SUB, CLOUD FUNCTIONS, CLOUD DATASTORE, CLOUD CDN, STACKDRIVER LOGGING, STACKDRIVER MONITORING

Version 2.0
Last modified: 2017-09-14



Google Cloud

## Build for the cloud

Global Reach

Scalability and
High Availability

Security



Google Cloud

Applications that run in the cloud must be built for global reach, scalability and high availability, and security.

- Global Reach: Your application should be responsive and accessible to users across the world.
- Scalability and High Availability: Your application should be able to handle high traffic volumes reliably. The application architecture should leverage the capabilities of the underlying cloud platform to scale elastically in response to changes in load.
- Security: Your application and the underlying infrastructure should implement security best practices. Depending on the use case, you might be required to isolate you user data in a specific region for security and compliance reasons.

Images:
https://pixabay.com/en/globe-earth-planet-continents-147715/

# Implement best practices to build scalable, more secure, and highly available applications






In this presentation, you will learn best practices related to code and environment management, design and development, scalability and reliability, and migration.

## Manage your application's code and environment

Code Repository

Dependency Management

Configuration Settings



Google Cloud

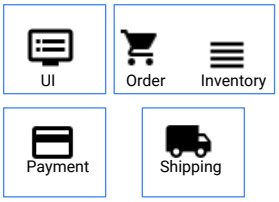
Implement the following best practices to manage your application's code and environment:

- Store your application's code in a code repository in a version control system such as Git or Subversion. This will enable you to track changes to your source code and set up systems for continuous integration and delivery.
- Do not store external dependencies such as JAR files or external packages in your code repository. Instead, depending on your application platform, explicitly declare your dependencies with their versions and install them using a dependency manager. For example, for a Node.js application, you can declare your application dependencies in a package.json file and later install them using the npm install command.
- Separate your application's configuration settings from your code. Do not store configuration settings as constants in your source code. Instead, specify configuration settings as environment variables. This enables you to easily modify settings between development, test, and production environments.

## Consider implementing microservices


Monolithic application

- Codebase becomes large.
- Packages have tangled dependencies.




Microservices

- Service boundaries match business boundaries.
- Codebase is modular.
- Each service can be independently updated, deployed, and scaled.



Google Cloud

Instead of implementing a monolithic application, consider implementing or refactoring your application as a set of microservices.

In a monolithic application, the codebase becomes bloated over time. It can be difficult to determine where code needs to be changed. Packages or components of the application can have tangled dependencies. The entire application needs to be deployed and tested even if a change is made to a small part of the codebase. This increases the effort and risk when making feature changes and bug fixes.

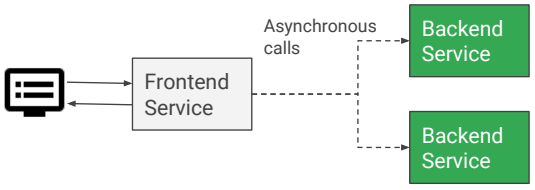
Microservices enable you to structure your application components in relation to your business boundaries. The codebase for each service is modular. It is easy to determine where code needs to be changed. Each service can be updated and deployed independently without requiring the customers to change simultaneously. Each service can be scaled independently depending on load.

Make sure to evaluate the costs and benefits of optimizing and converting a monolithic application into one that uses a microservices architecture.

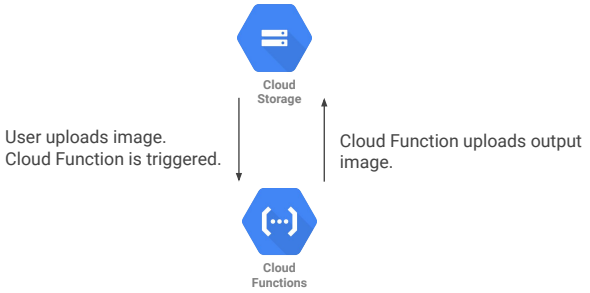
Images:
https://material.io/icons/

## Perform asynchronous operations

Keep UI responsive; perform backend operations asynchronously.




Use event-driven processing.





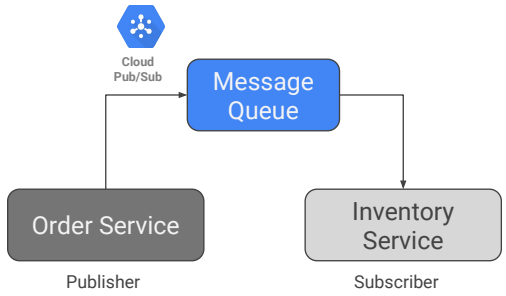

Google Cloud

Remote operations can have unpredictable response times and can make your application seem slow. Keep the operations in the user thread at a minimum. Perform backend operations asynchronously.

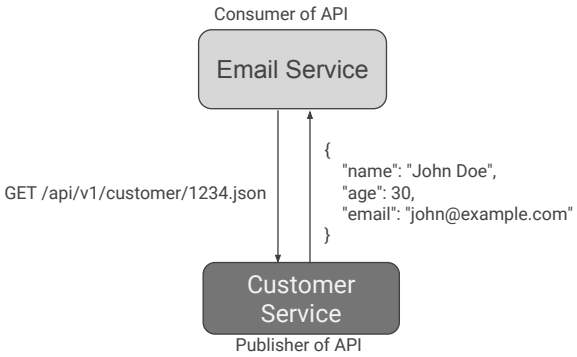
Use event-driven processing where possible. For example, if your application processes images that are uploaded by a user, you can use a Google Cloud Storage bucket to store the uploaded images. You can then implement a Google Cloud Function that is triggered whenever a new image is uploaded. The Google Cloud Function processes the image and uploads the results to a different Cloud Storage location.

## Design for loose coupling

Publishers and subscribers are loosely coupled.




Consumers of HTTP APIs should bind loosely with publisher payloads.






Google Cloud

Design application components so that they are loosely coupled at runtime. Tightly coupled components can make an application less resilient to failures, spikes in traffic, and changes to services.

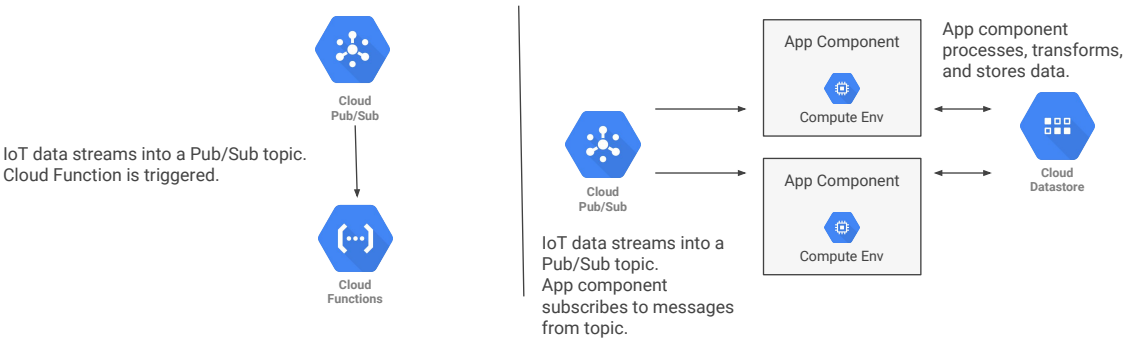
An intermediate component such as a message queue can be used to implement loose coupling, perform asynchronous processing, and buffer requests in case of spikes in traffic. You can use a Cloud Pub/Sub topic as a message queue. Publishers can publish messages to the topic, and subscribers can subscribe to messages from this topic.

In the context of HTTP API payloads, consumers of HTTP APIs should bind loosely with the publishers of the API. In the example, the Email service retrieves information about each customer from the Customer service. The Customer service returns the customer's name, age, and email address in its payload. To send an email, the Email service should only reference the name and email fields in the payload. It should not attempt to bind with all fields in the payload. This method of loosely binding fields will enable the publisher to evolve the API and add fields to the payload in a backwards-compatible manner.

# Implement stateless components for scalability

Worker pattern

Workers perform compute tasks without sharing state.
Workers can scale up and down reliably.






Google Cloud

Implement application components so that they do not store state internally or access a shared state. Accessing a shared state is a common bottleneck for scalability. Design each application component so that it focuses on compute tasks only. This approach enables you to use a Worker pattern to add or remove additional instances of the component for scalability. Application components should start up quickly to enable efficient scaling and shut down gracefully when they receive a termination signal.

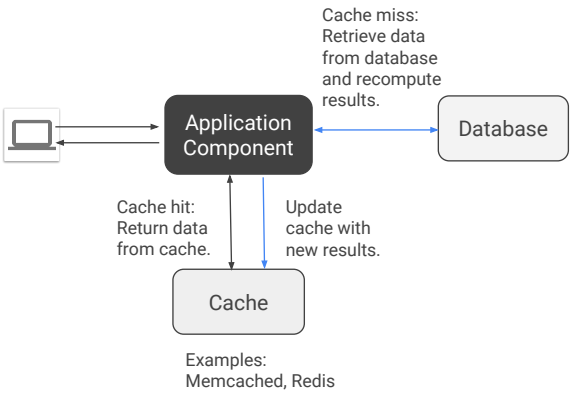
For example, if your application needs to process streaming data from IoT devices, you can use a Google Cloud Pub/Sub topic to receive the data. You can then implement a Google Cloud Function that is triggered whenever a new piece of data comes in. The Google Cloud Function can process, transform, and store the data.

Alternatively, your application can subscribe to the Pub/Sub topic that receives the streaming data. Multiple instances of your application can spin up and process the messages in the topic and split the workload. These instances can automatically be shut down when there are few messages to process. To enable elastic scaling, you can use any compute environment, such as Google Compute Engine with Google Cloud Load Balancing, Google Container Engine, or Google App Engine.

With either approach, you do not have to develop code to manage concurrency or scaling. Your application scales automatically depending on the workload.

## Cache content

Cache application data.




Cache frontend content.

Cloud CDN

- Cache load-balanced frontend content that comes from Compute Engine VM instance groups.
- Cache static content that is served from Cloud Storage.



Google Cloud

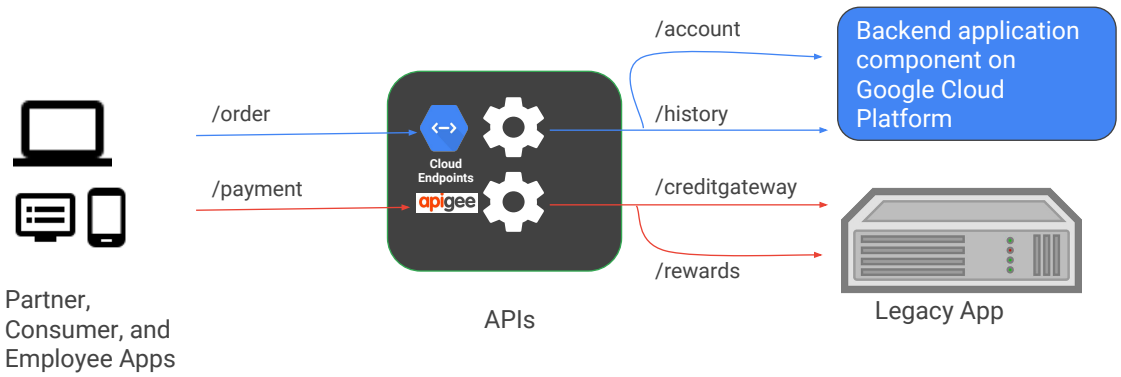
Caching content can improve application performance and lower network latency.

Cache application data that is frequently accessed or that is computationally intensive to calculate each time.

When a user requests data, the application component should check the cache first. If data exists in the cache (TTL has not expired), the application should return the previously cached data. If the data does not exist in the cache or has expired, the application should retrieve the data from backend data sources and recompute results as needed. The application should also update the cache with the new value.

In addition to caching application data in a cache such as Memcached or Redis, you can also use a content delivery network to cache web pages. Cloud Content Delivery Network can cache load-balanced frontend content that comes from Compute Engine VM instance groups or static content that is served from Cloud Storage. For more information about using Cloud CDN, see https://cloud.google.com/cdn/docs/overview.

## Implement API gateways to make backend functionality available to consumer applications






Google Cloud

Implement API gateways to make backend functionality available to consumer applications.

You can use Google Cloud Endpoints to develop, deploy, protect and monitor APIs based on the OpenAPI specification. Further, the API for your application can run on backends such as Google App Engine, Google Container Engine, or Compute Engine.

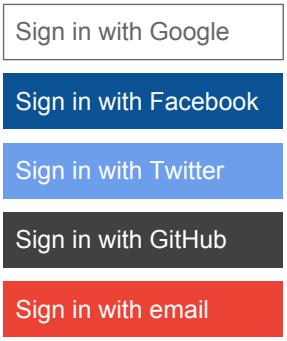
If you have legacy applications that cannot be refactored and moved to the cloud, consider implementing APIs as a facade or adapter layer. Each consumer can then invoke these modern APIs to retrieve information from the backend instead of implementing functionality to communicate using outdated protocols and disparate interfaces.

Using the Apigee API platform, you can design, secure, analyze, and scale your APIs for legacy backends. For more information about the Apigee API platform, see:

- Getting Started: http://docs.apigee.com/api-services/content/what-apigee-edge
- Training: http://academy.apigee.com/index.php

Images:
https://pixabay.com/en/computer-network-router-server-159829/
https://material.io/icons/

# Use federated identity management




Firebase Authentication

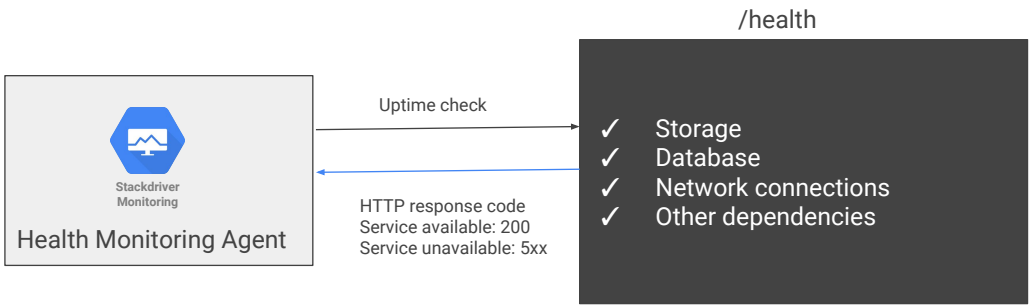
Authenticate users by using external identity providers.





Delegate user authentication to external identity providers such as Google, Facebook, Twitter, or GitHub. This will minimize your effort for user administration. With federated identity management, you do not need to implement, secure, and scale a proprietary solution for authenticating users.

## Implement health-check endpoints






Google Cloud

It is important to monitor the status of your application and services to ensure that they are always available and performing optimally. The monitoring data can be used to automatically alert operations teams as soon as a system begins to fail. Operations teams can then diagnose and address the issue promptly.

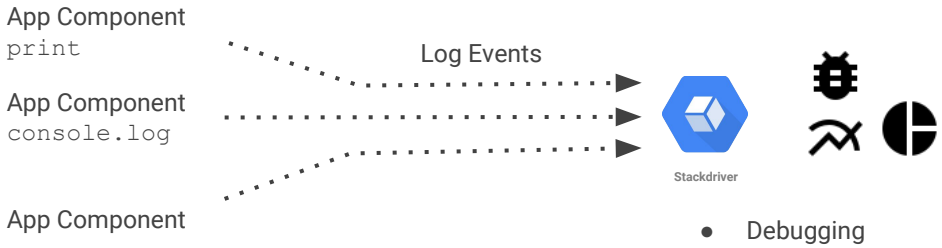
Implement a health-check endpoint for each service. The endpoint handler should check the health of all dependencies and infrastructure components required for the service to function properly. For example, the endpoint handler can check the performance and availability of storage, database, and network connections required by the service. The endpoint handler should return an HTTP response code of 200 for a successful health check. If the health check fails, the endpoint handler can return 503 or a more specific response code depending on the error.

Evaluate which dependencies are critical enough to result in a health-check failure. For example, consider a UI service that displays recommendations in one section of the web page. The UI is designed to degrade gracefully (hide the recommendations section) if the recommendations engine is down.  With this design, if the recommendation engine fails, you can report the failure on the health status web page, but still return a successful health check for the UI service. This approach ensures that your application continues to receive traffic even if some non-critical dependencies are unavailable.

A health monitoring agent such as a load balancer or Stackdriver Monitoring can periodically send requests to the health-check endpoint. Load balancers use health

checks to determine if an application instance is healthy and is capable of receiving traffic. If the health check fails beyond the threshold that you have defined, Stackdriver Monitoring can send you an alert automatically.

## Set up logging and monitor your application's performance




- Debugging
- Error reporting
- Tracing
- Logs-based metrics
- Monitoring



Google Cloud

Treat your logs as event streams. Logs constitute a continuous stream of events that keep occurring as long as the application is running. Do not manage log files in your application. Instead, write to an event stream such as `stdout` and let the underlying infrastructure collate all events for later analysis and storage. With this approach, you can set up logs-based metrics and trace requests across different services in your application.

With Google Stackdriver, you can debug your application, set up error reporting, set up logging and logs-based metrics, trace requests across services, and monitor applications running in a multi-cloud environment.

## Handle transient and long-lasting errors gracefully

Transient errors:
Retry with exponential backoff.

Service availability errors:
Implement a circuit breaker.



Google Cloud

When accessing services and resources in a distributed system, applications need to be resilient to temporary and long-lasting errors.

Resources can sometimes become unavailable due to transient network errors. In this case, applications should implement retry logic with exponential backoff and fail gracefully if the errors persist. The Google Cloud Client Libraries retry failed requests automatically.

When errors are more long-lasting, the application should not waste CPU cycles attempting to retry the request. In this case, applications should implement a circuit breaker and handle the failure gracefully.

For errors that are propagated back to the user, consider degrading the application gracefully instead of explicitly displaying the error message. For example, if the recommendation engine is down, consider hiding the product recommendations area on the page instead of displaying error messages every time the page is displayed.

For information about exceptions related to Cloud Client Libraries for Python, see https://googlecloudplatform.github.io/google-cloud-python/stable/core/modules.html#module-google.cloud.exceptions.

## Consider data sovereignty and compliance requirements




Google Cloud

Some regions and industry segments have strict compliance requirements for data protection and consumer privacy. For example, European Union's Data Protection Directive regulates the processing of personal data. The directive specifies a number of requirements that companies must meet around protecting personal data.

Review the data protection requirements for the industry segments and the regions where your users and services will be located. For more information about security and compliance in Google Cloud Platform, see:

- Google Cloud Platform Security: https://cloud.google.com/security/
- Privacy and Compliance: https://cloud.google.com/security/compliance

Images:
https://cloud.google.com/security/

## Perform high availability testing and develop disaster recovery plans

In addition to functional and performance testing, perform high-availability testing and develop disaster recovery plans.


- Identify failure scenarios.
- Create disaster recovery plans (people, processes, tools).
- Perform tabletop tests.


- Perform canary testing and blue/green deployments.
- Validate your disaster recovery plan.

Example failure scenarios:

- Connectivity failure
- On-premises data center or other cloud-provider failure
- GCP zonal or regional failure
- Deployment rollback
- Data corruption caused by network or application issues



Google Cloud

In addition to executing functional and performance testing, execute high-availability testing. Such testing will enable you to develop robust disaster recovery plans and perform disaster recovery practice exercises regularly.

Identify failure scenarios and create disaster recovery plans that identify the people, processes, and tools for disaster recovery. Initially, you can perform tabletop tests. These are tests in which teams discuss how they would respond in failure scenarios but do not perform any real actions. This type of test encourages teams to discuss what they would do in unexpected situations.

Then, simulate failures in your test environment. After you understand the behavior of your application under failure scenarios, address any problems, and refine your disaster recovery plan. Then, test the failure scenarios in your production environment. You can perform canary testing or blue/green deployments on your production environment. Consider scheduling such testing during a maintenance window or during off-peak hours to minimize impact.

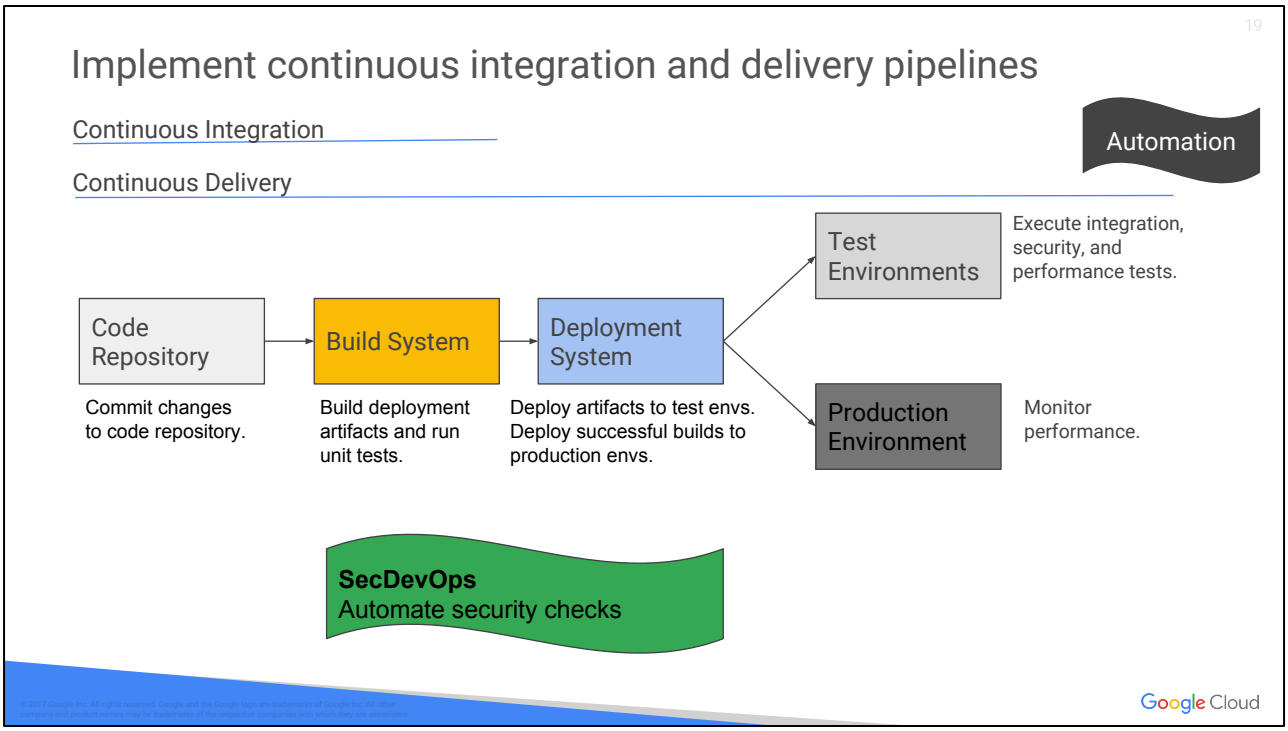
Consider failure scenarios such as the following:

- Connectivity failure: When VPN and other such connections between the on-premises data center and GCP fail, traffic should be rerouted through other redundant routes.
- On-premises data center or other cloud-provider failure: The application and its environment should continue to function effectively even in the case of an

- on-premises data center outage or an outage with another cloud provider.
- GCP zonal or regional failure: In case of a zone failure or region failure, make sure that traffic is routed to alternate zones or regions. Zonal failures are uncommon, and regional failures are extremely rare.
- Deployment rollback: In some cases, you might have to roll back a software deployment to an older, stable version. Test the rollback procedure.
- Data corruption caused by network or application issues:  It is important to have a process to restore data from backups in case of data corruption caused by network or application issues.

In each of these failure scenarios, verify the following:

- The appropriate people were notified about the failure in a timely manner.
- Traffic was rerouted using redundant routes.
- Data was not corrupted during the failure or recovery process.
- All services were restored in a timely manner.




Implement a strong DevOps model with automation to enable continuous integration and delivery (CI/CD). Automation helps you increase release velocity and reliability. With a robust CI/CD pipeline, you can test and roll out changes incrementally instead of making big releases with multiple changes. This approach enables you to lower the risk of regressions, debug issues quickly, and roll back to the last stable build if necessary.

In a continuous integration system, developers commit code into a code repository such as Git. A build system such as Jenkins automatically detects commits to the repository, triggers builds, and runs unit tests. The build system produces deployment artifacts for all required runtime environments.

In a continuous delivery system, a deployment system such as Spinnaker automatically triggers the deployment of builds to test environments. You can automatically execute integration, security, and performance tests and then deploy successful builds to your production environment. When rolling out builds to the production environment, consider performing canary testing or blue/green deployments to make sure that any unexpected issues do not affect a large number of users.

You can set up performance metrics and alerts to monitor the application's performance in production.

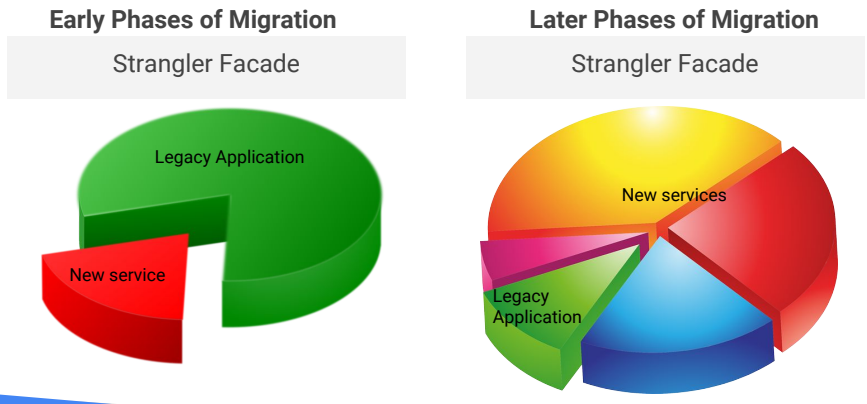
It is important to consider security throughout the continuous integration and delivery

process. With a SecDevOps approach you can automate security checks such as the following:

- Confirm that most recent and secure versions of third-party software and dependencies are used
- Scan code for security vulnerabilities
- Confirm that resources have permissions based on principles of least privilege
- Confirm that logs and events sent by services are sent to a central location
- Detect errors in production and roll back to the last stable build

## Use the strangler pattern to re-architect applications

Strangler pattern: Incrementally replace components of the old application with new services.






Google Cloud

Consider using the Strangler pattern when re-architecting and migrating large applications. In the early phases of migration, you might replace smaller components of the legacy application with newer application components or services. You can incrementally replace more features of the original application with new services. A Strangler facade can receive requests and direct them to the old application or new services. As your implementation evolves, the legacy application is “strangled” by the new services and no longer required.

This approach minimizes risk by enabling you to learn from each service implementation without affecting business-critical operations.

The term “strangler” comes from strangler vines. The vines seed and start growing on the upper branches of fig trees, gradually envelope the tree, and finally kill the tree that acted as their host.

Images:
https://pixabay.com/en/pie-chart-diagram-3d-graph-pie-153903/
https://pixabay.com/en/pie-chart-diagram-statistics-parts-149727/
https://pixabay.com/en/strangler-fig-jungle-trees-forest-453715/