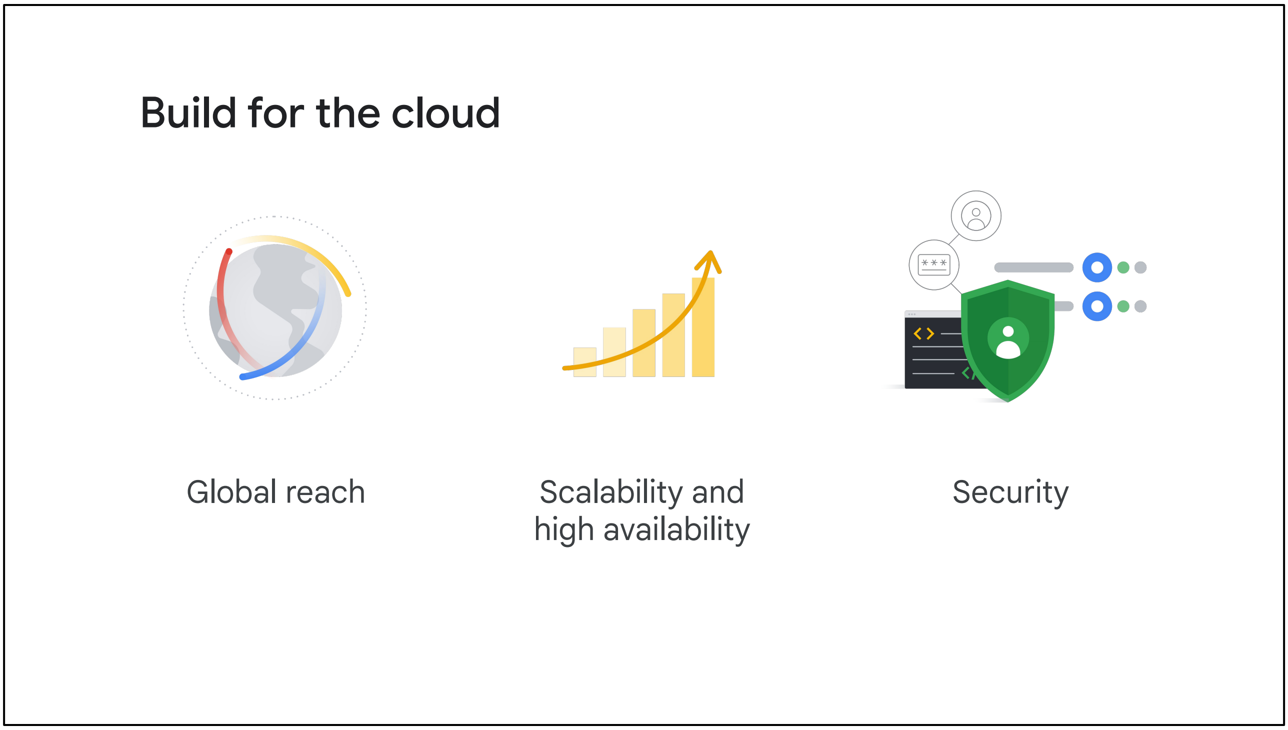
Developing Applications with Google Cloud: Foundations

Module 1: Best Practices for
Cloud Application Development

Welcome to Developing Applications with Google Cloud: Foundations, Module 1: Best Practices for Cloud Application Development.

Before we discuss the details of Google Cloud application development, we look at some best practices for application development in the cloud.



Build for the cloud

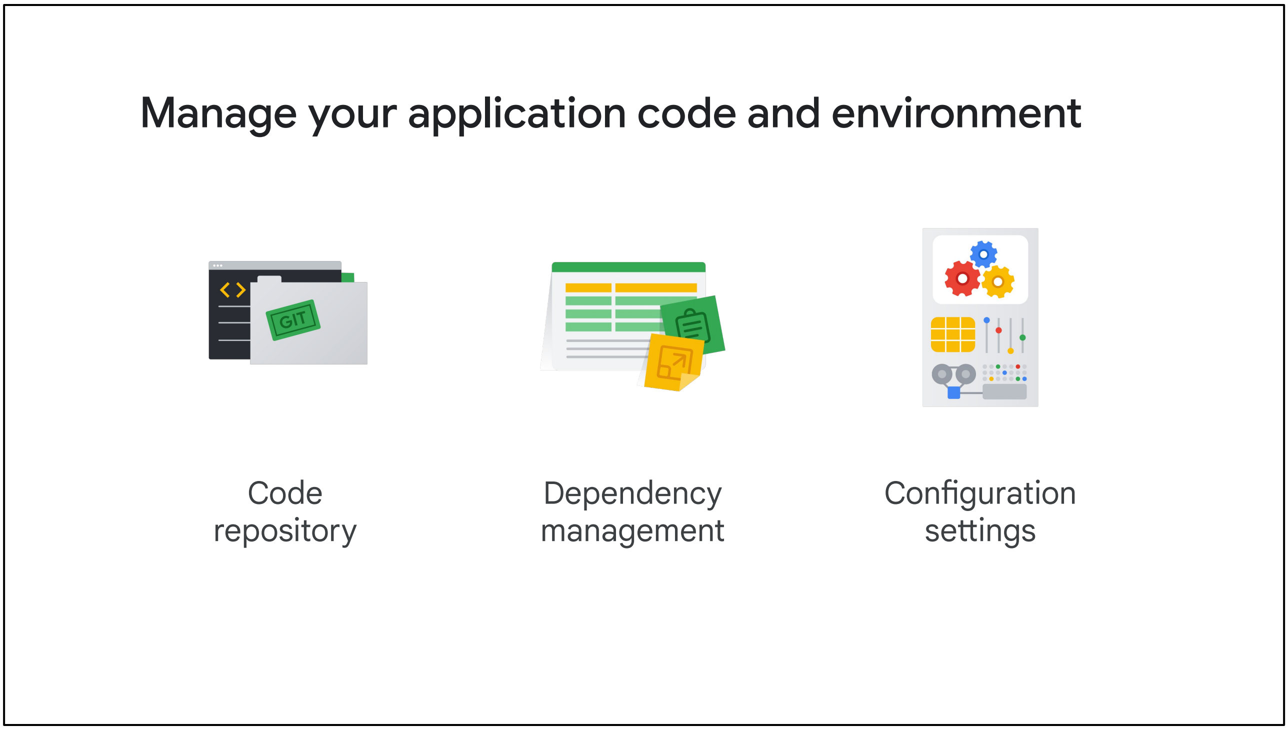
Global reach

Scalability and high availability

Security

Applications that run in the cloud must be built to handle:

- 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 use 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 your user data in a specific region for security and compliance reasons.



Manage your application code and environment

Code repository

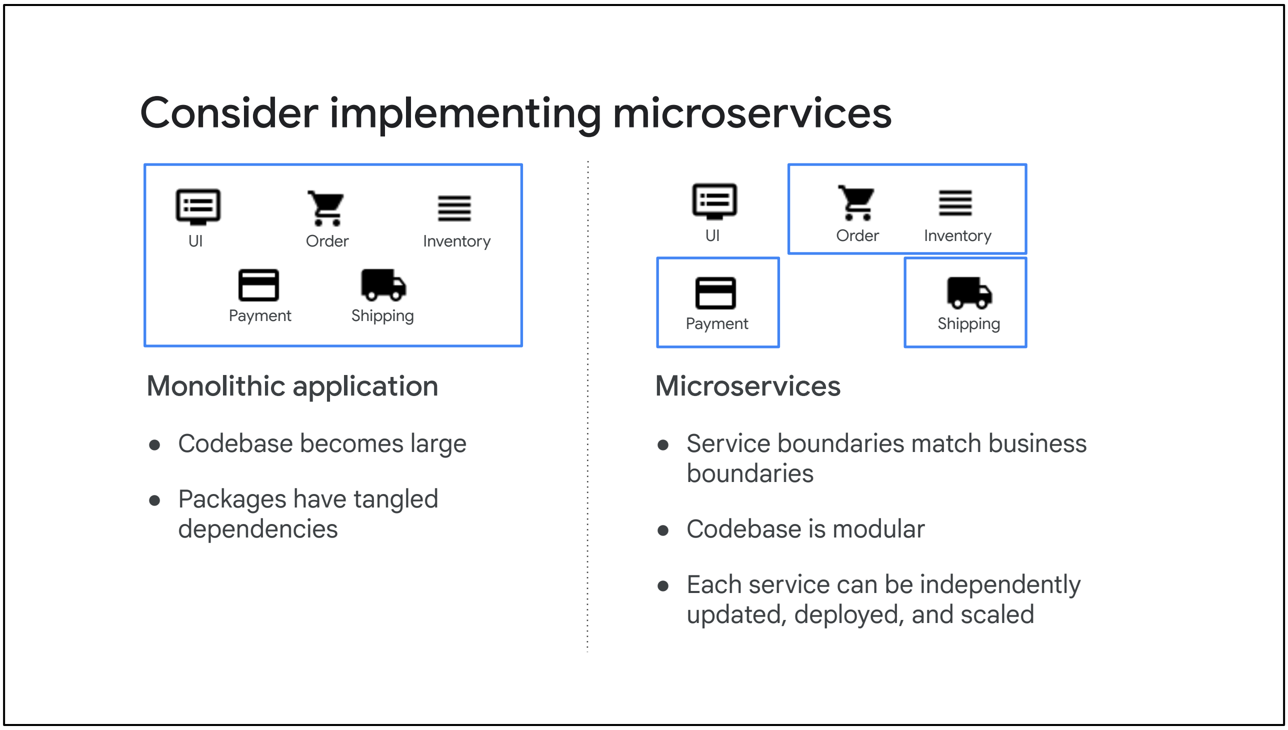
Dependency management

Configuration settings

Here are some best practices for managing your application code and environment. First, store your code in a code repository in a version control system such as Git. Using a code repository will enable you to track changes to your source code and set up systems for continuous integration and delivery.

Second, 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 by using a dependency manager. For example, in a Node.js application, you can declare your application dependencies in a package.json file and later install them by using the npm install command.

Third, separate your configuration settings from your code. Do not store configuration settings as constants in your source code. Instead, specify configuration settings as environment variables. Passing configuration by using environment variables lets you modify settings between development, test, and production environments, which ensures that the tested code is used in production.



Consider implementing microservices

Monolithic application

- Codebase becomes large
- Packages have tangled dependencies

Microservices

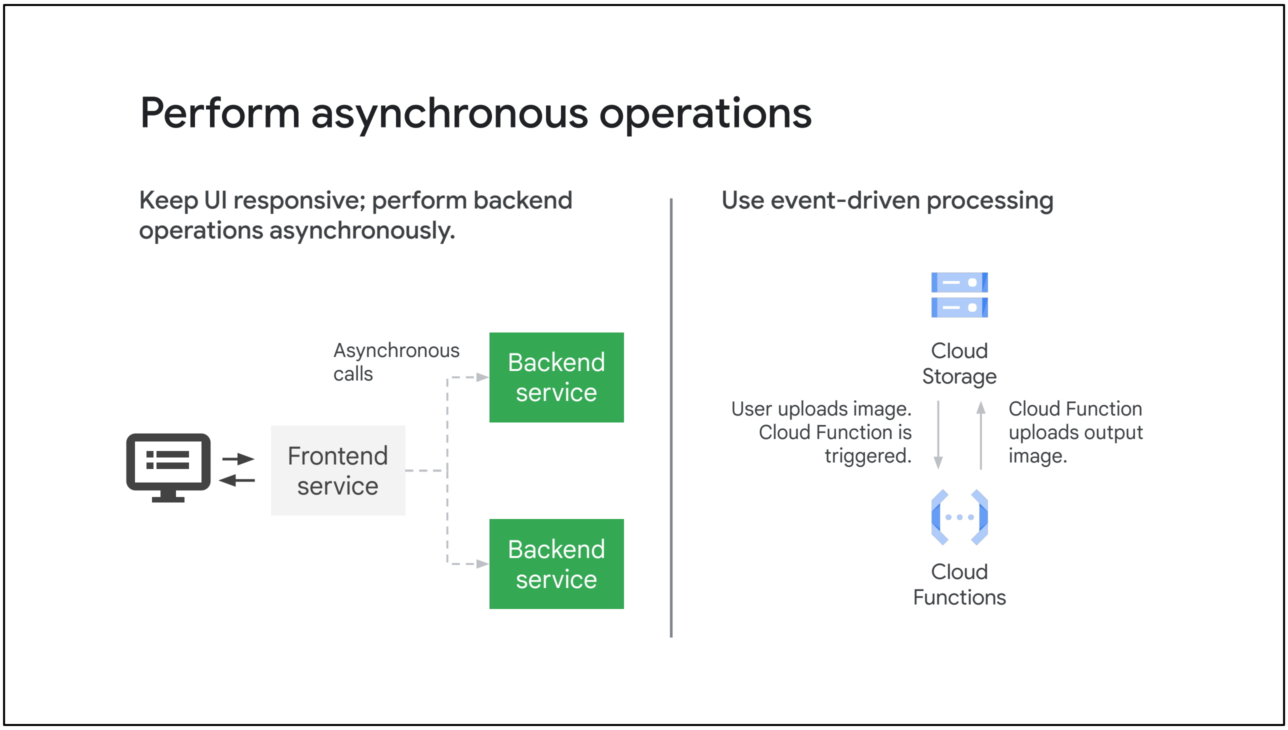
- Service boundaries match business boundaries
- Codebase is modular
- Each service can be independently updated, deployed, and scaled

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

Most applications are changed significantly over time. In a monolithic application, the codebase may become bloated. It can be difficult to determine all code that needs to change when implementing a new feature. Packages or components of the application can have tangled dependencies. The entire application must be deployed and tested even if a change is only made to a small part of the codebase. Monolithic applications increase the effort and risk when feature changes and bug fixes are done.

Microservices enable you to structure your application components in relation to your business boundaries. Refactoring a monolithic application into microservices may require significant time and effort, but the benefits gained can be worth the cost. The codebase for each microservice is modular. It's easy to determine where code needs to be changed. Each service can be updated, tested, and deployed independently without requiring its customers to change simultaneously. Each service can be scaled independently, depending on the load.

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



Perform asynchronous operations

Keep UI responsive; perform backend operations asynchronously.

Asynchronous calls

Frontend service

Backend service

Backend service

Use event-driven processing

Cloud Storage

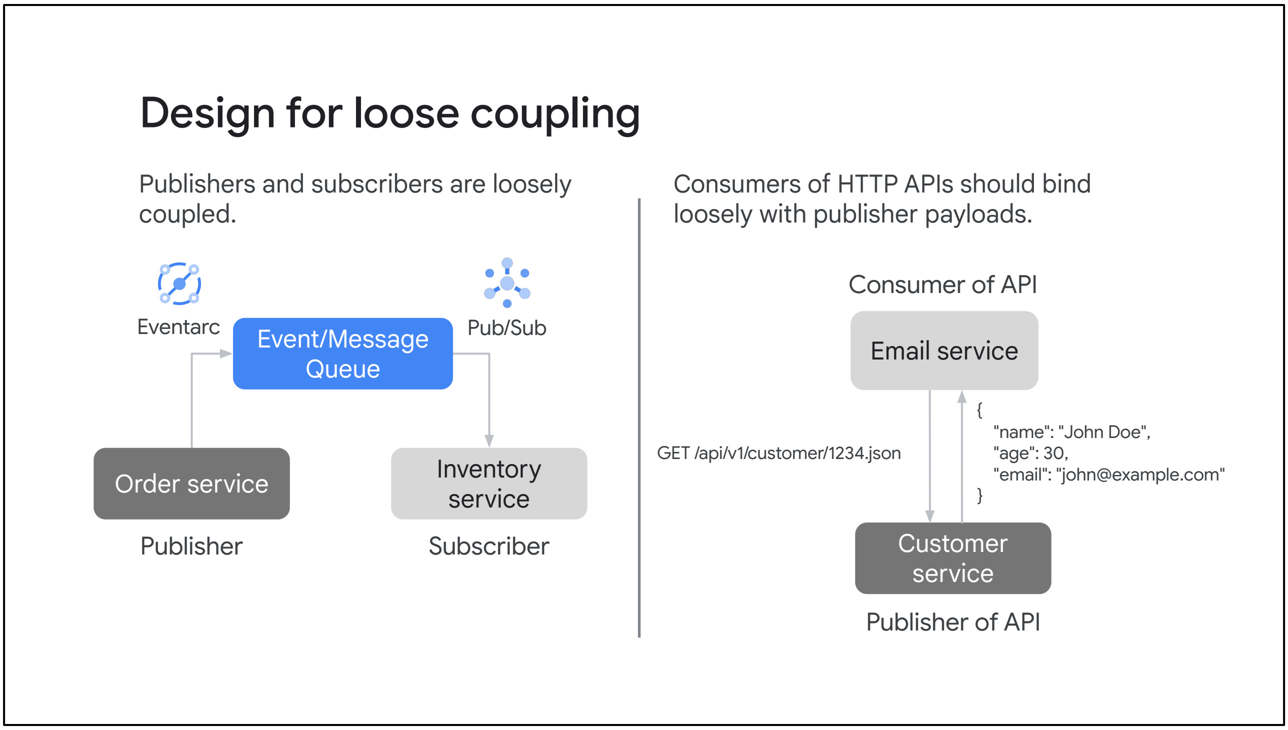
User uploads image. Cloud Function is triggered.

Cloud Function uploads output image.

Cloud Functions

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

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

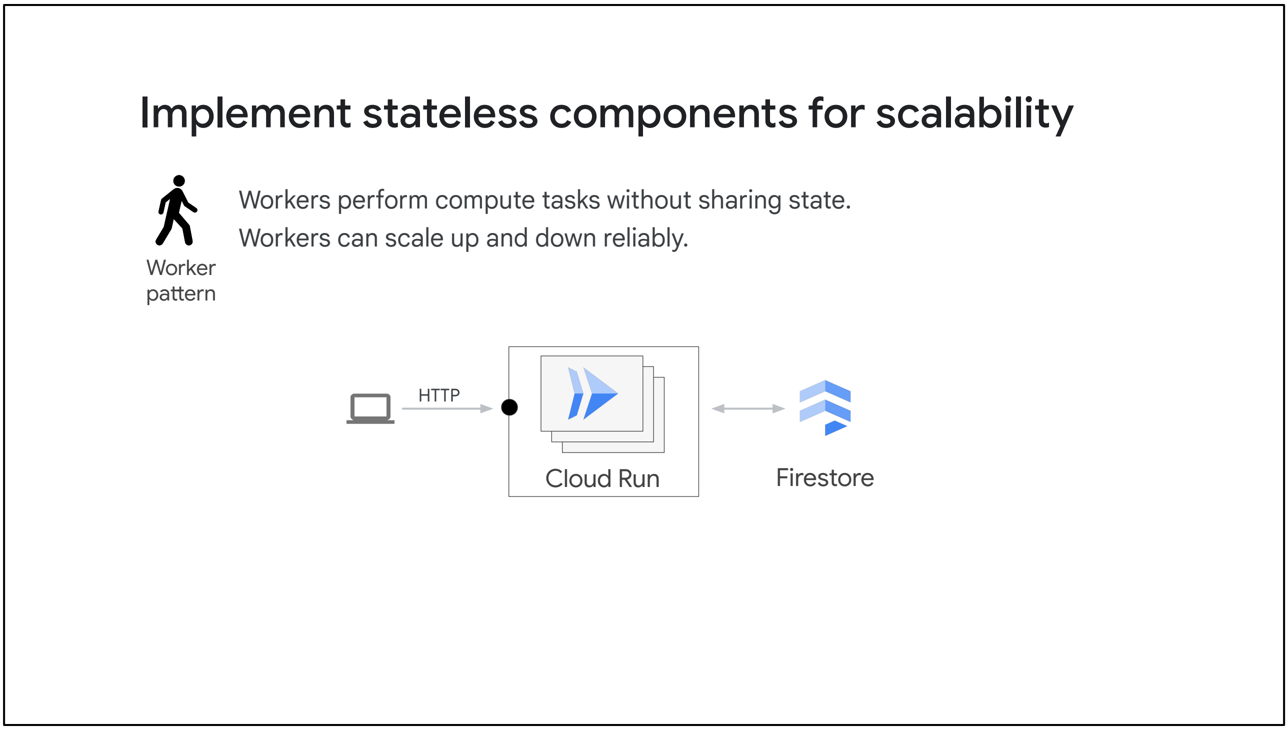
GET /api/v1/customer/1234.json

```
{
  "name": "John Doe",
  "age": 30,
  "email": "john@example.com"
}
```

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 event or message queue can be used to implement loose coupling, perform asynchronous processing, and buffer requests if traffic spikes. You can use an Eventarc trigger as an event queue or a Pub/Sub topic as a message queue. The order and inventory services are loosely coupled, which allows them to operate independently.

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 name, age, and email address of the customer 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 allows the publisher to evolve the API and add fields to the payload in a backward compatible manner.



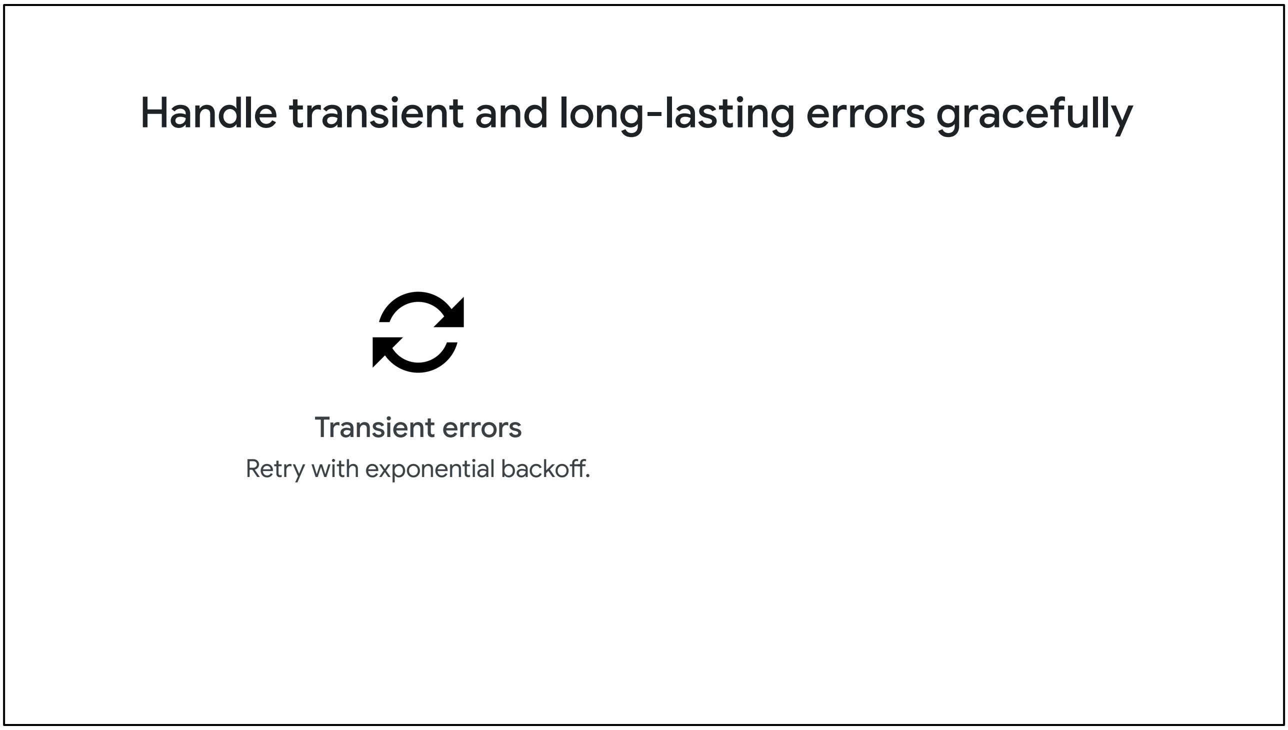
Implement stateless components for scalability

Worker pattern

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

Implement application components so that they do not store state internally or share 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 allows 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 then shut down gracefully when they receive a termination signal.

For example, if traffic to your application can vary significantly, you can use Cloud Run for your application, and scale capacity based on traffic. The Cloud Run services process the incoming requests but don't store or share state, so they can be shut down easily when traffic is reduced. Data should be persisted in a separate database like Firestore.



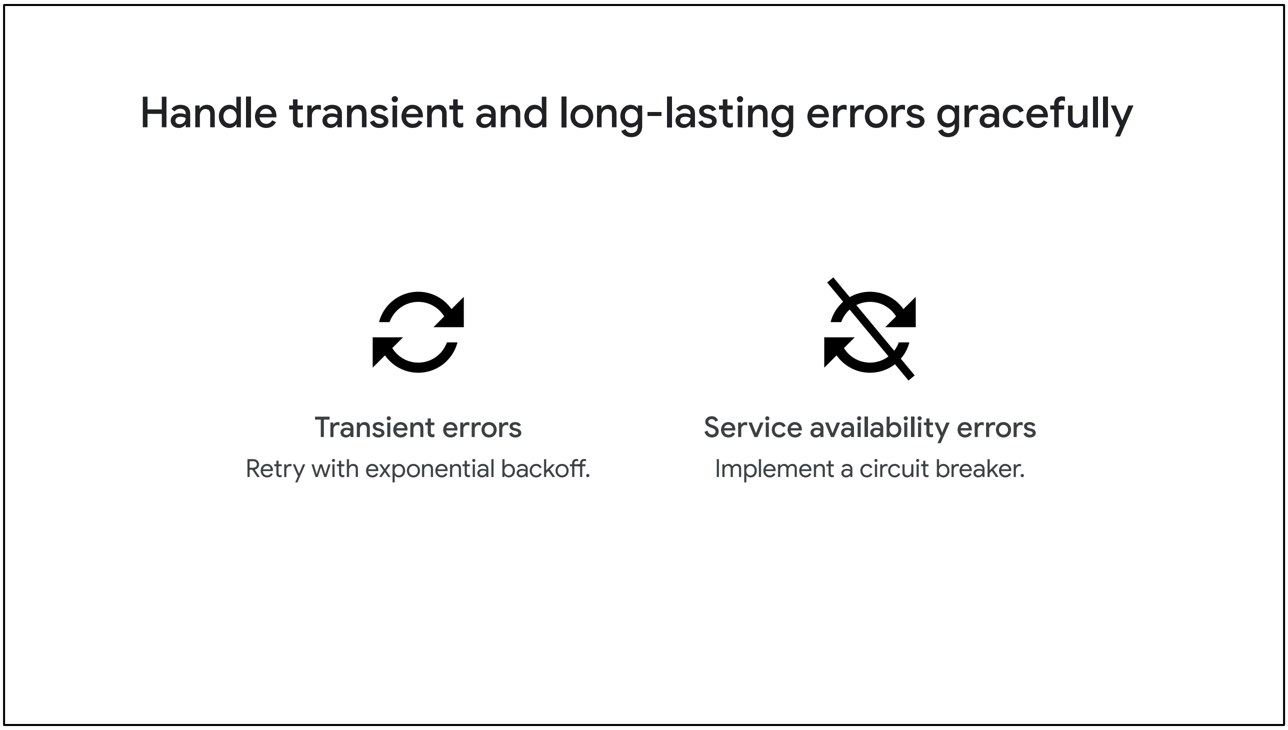
Handle transient and long-lasting errors gracefully

Transient errors

Retry with exponential backoff.

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. Exponential backoff helps ensure that applications don't make the problem worse by overloading the backend or network. The Cloud Client Libraries retry failed requests automatically.



Handle transient and long-lasting errors gracefully

Transient errors

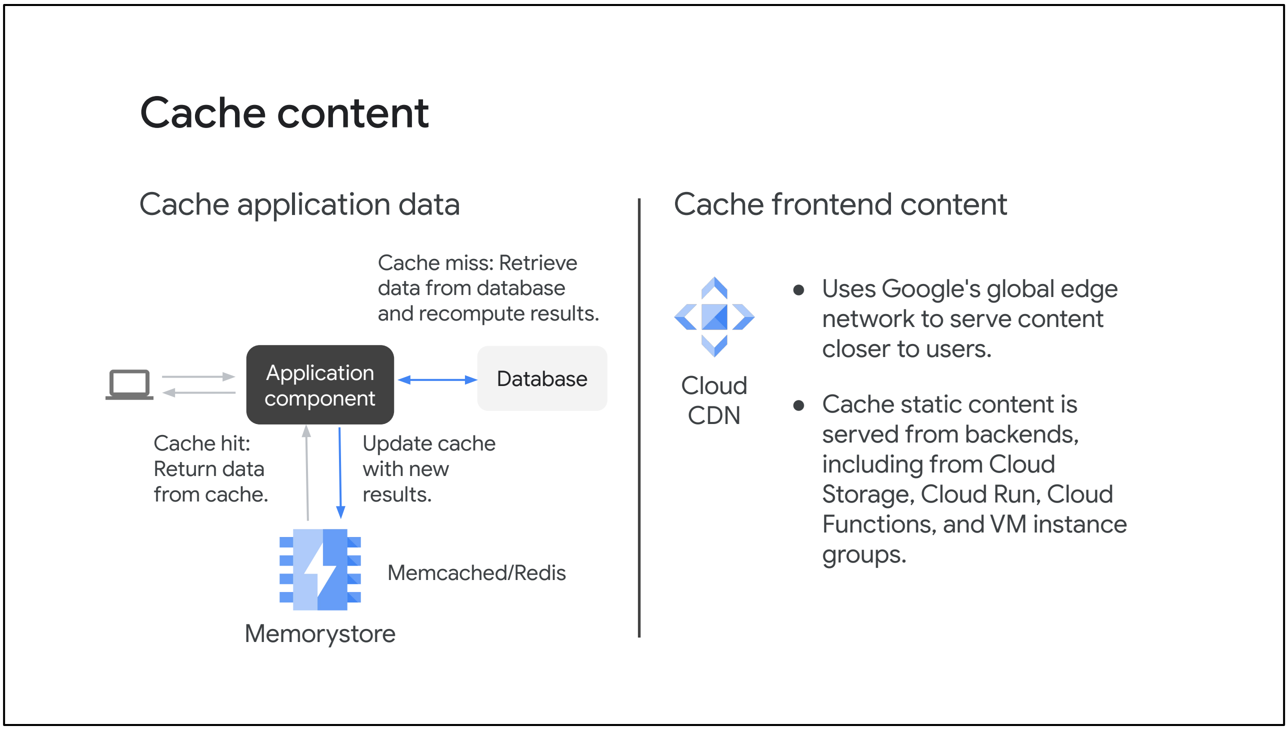
Retry with exponential backoff.

Service availability errors

Implement a circuit breaker.

When services are down with long-lasting errors, the application should not generate traffic or 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 recommendations section instead of displaying error messages every time the page is displayed.



Cache content

Cache application data

Cache miss: Retrieve data from database and recompute results.

Application component

Database

Cache hit: Return data from cache.

Update cache with new results.

Memcached/Redis

Memorystore

Cache frontend content

Cloud CDN

- Uses Google's global edge network to serve content closer to users.
- Cache static content is served from backends, including from Cloud Storage, Cloud Run, Cloud Functions, and VM instance groups.

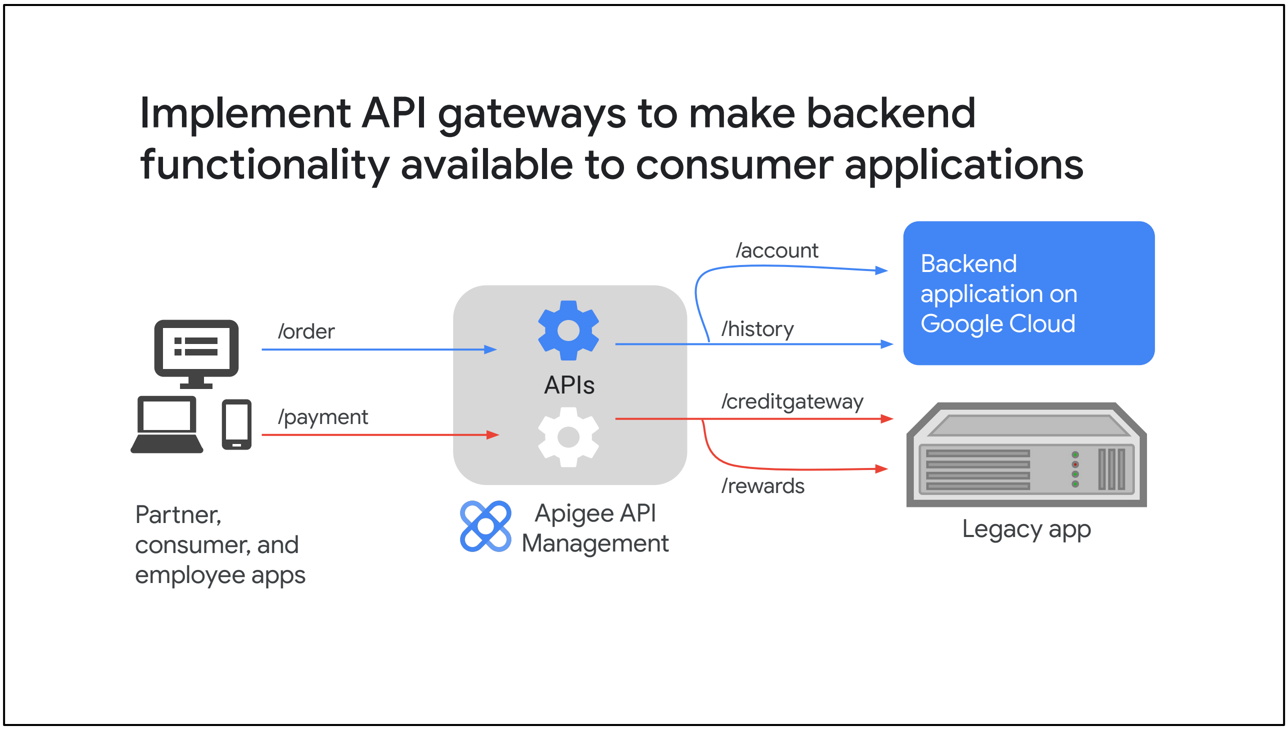
Caching content can improve application performance and lower network latency.

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, the application should return the previously cached data. If the data is not in the cache or has expired, the application should retrieve the data from backend data sources and recompute results. The application should also update the cache with the new value. Memorystore is Google Cloud's fully managed in-memory service for caching using Redis or Memcached.

In addition to caching application data in a cache, you can also use a content delivery network, or CDN, to cache web content. Cloud CDN uses Google's global edge network to serve content closer to users, which accelerates your websites and applications. Static content can be served from Cloud Storage buckets, services running on Cloud Run or Cloud Functions, or Compute Engine virtual machine instance groups.

[Memorystore: https://cloud.google.com/memorystore
Cloud CDN: https://cloud.google.com/cdn/docs/overview]



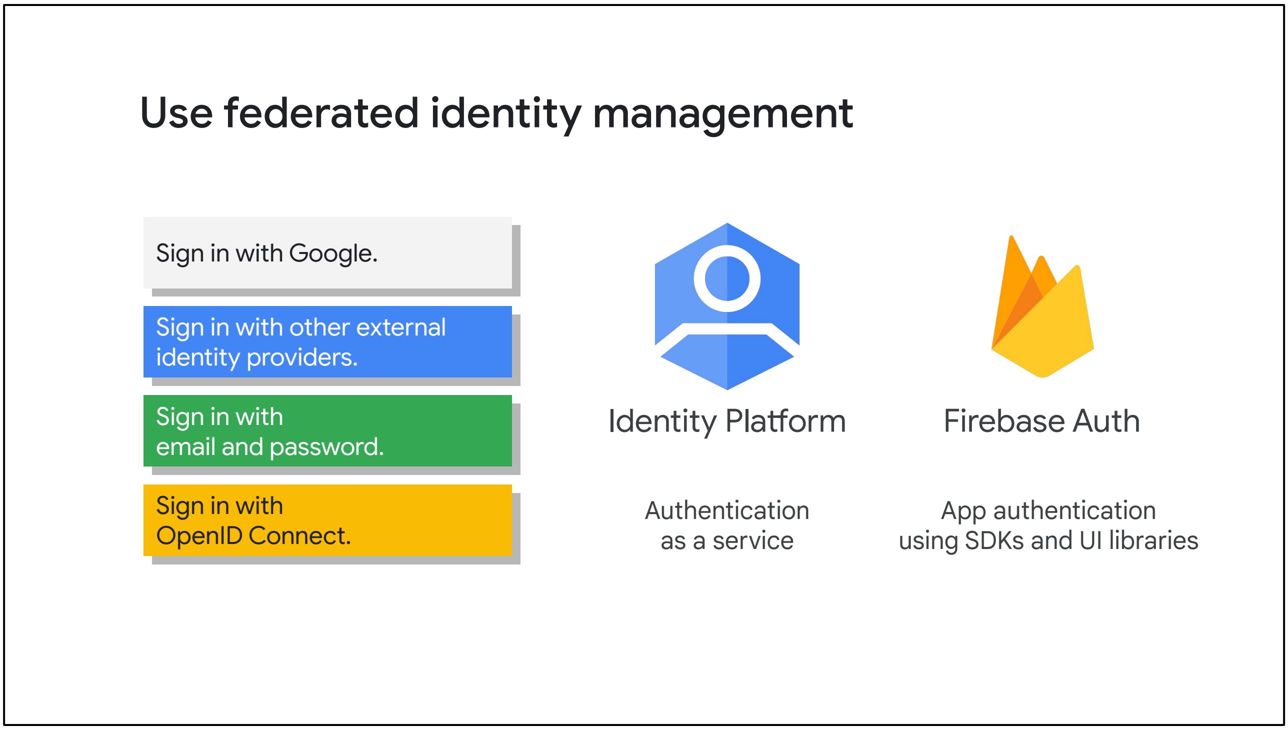
Implement API gateways to make backend functionality available to consumer applications

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

Apigee is a platform for developing and managing APIs. By fronting services with a proxy layer, Apigee acts as a facade for your backend service APIs, and provides security, rate limiting, quotas, analytics, and more.

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

[What is Apigee?
https://cloud.google.com/apigee/docs/api-platform/get-started/what-apigee]



Use federated identity management

Sign in with Google.

Sign in with other external identity providers.

Sign in with email and password.

Sign in with OpenID Connect.

Identity Platform

Authentication as a service

Firebase Auth

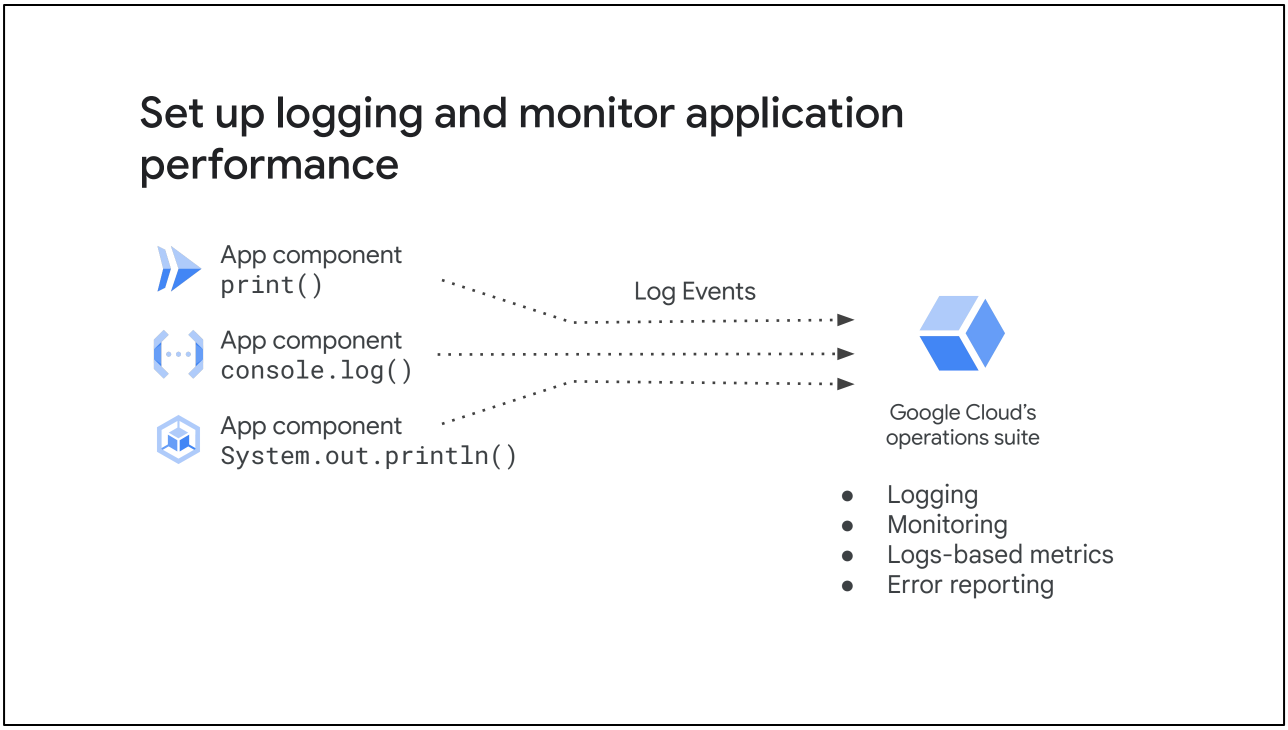
App authentication using SDKs and UI libraries

You can minimize your effort for user administration by delegating identity management. You can delegate user authentication to Google or other external providers like Facebook or GitHub.

Identity Platform provides a drop-in, customizable authentication service for user sign-up and sign-in. In addition to external providers, Identity Platform supports other authentication methods like email/password, SAML, OpenID Connect, and multi-factor authentication.

Firebase Authentication with Identity Platform provides backend services, easy-to-use SDKs, and ready-made UI libraries to authenticate users to your app.

With federated identity management, you do not need to implement, secure, and scale a proprietary solution for authenticating users.



Set up logging and monitor application performance

App component `print()`

App component `console.log()`

App component `System.out.println()`

Log Events

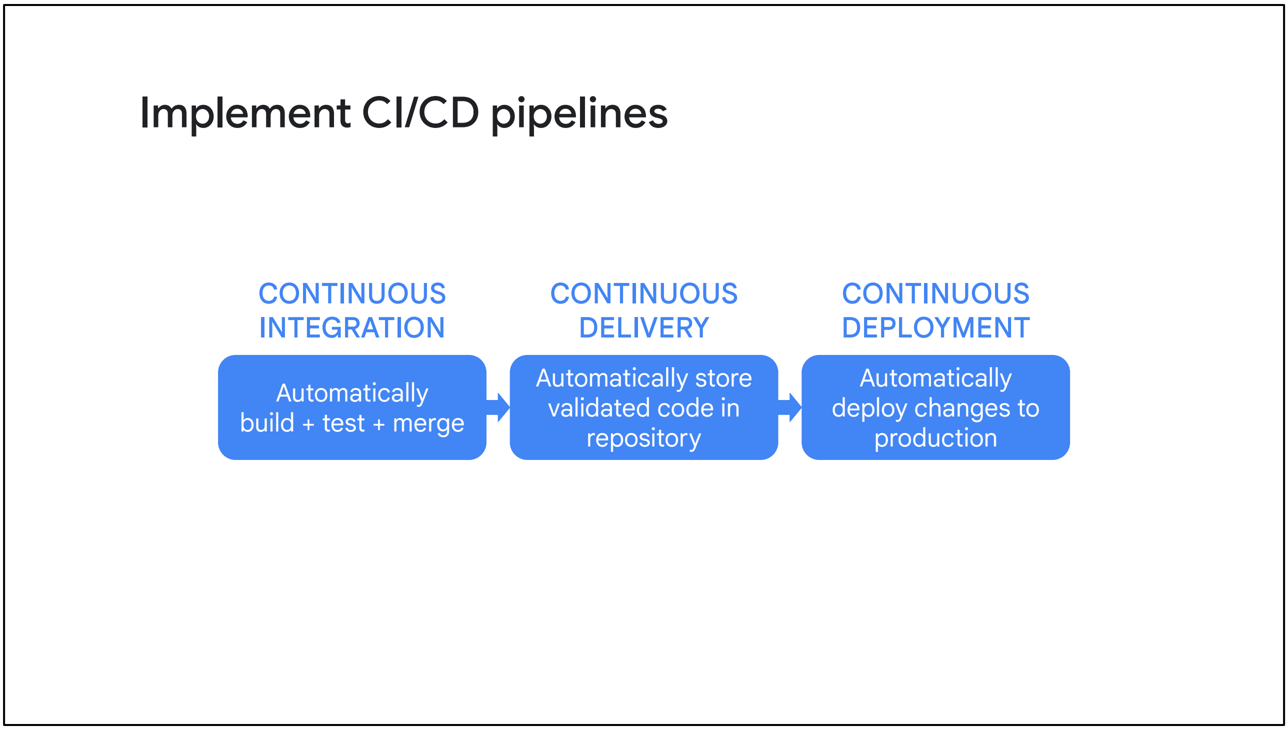
Google Cloud's operations suite

- Logging
- Monitoring
- Logs-based metrics
- Error reporting

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 like `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. Logging by writing to `stdout` works especially well for serverless compute options like Cloud Run and Cloud Functions.

With Google Cloud's operations suite, you can set up error reporting, logging and logs-based metrics, and monitor applications running in a multi-cloud environment.



Implement CI/CD pipelines

CONTINUOUS INTEGRATION

Automatically build + test + merge

CONTINUOUS DELIVERY

Automatically store validated code in repository

CONTINUOUS DEPLOYMENT

Automatically deploy changes to production

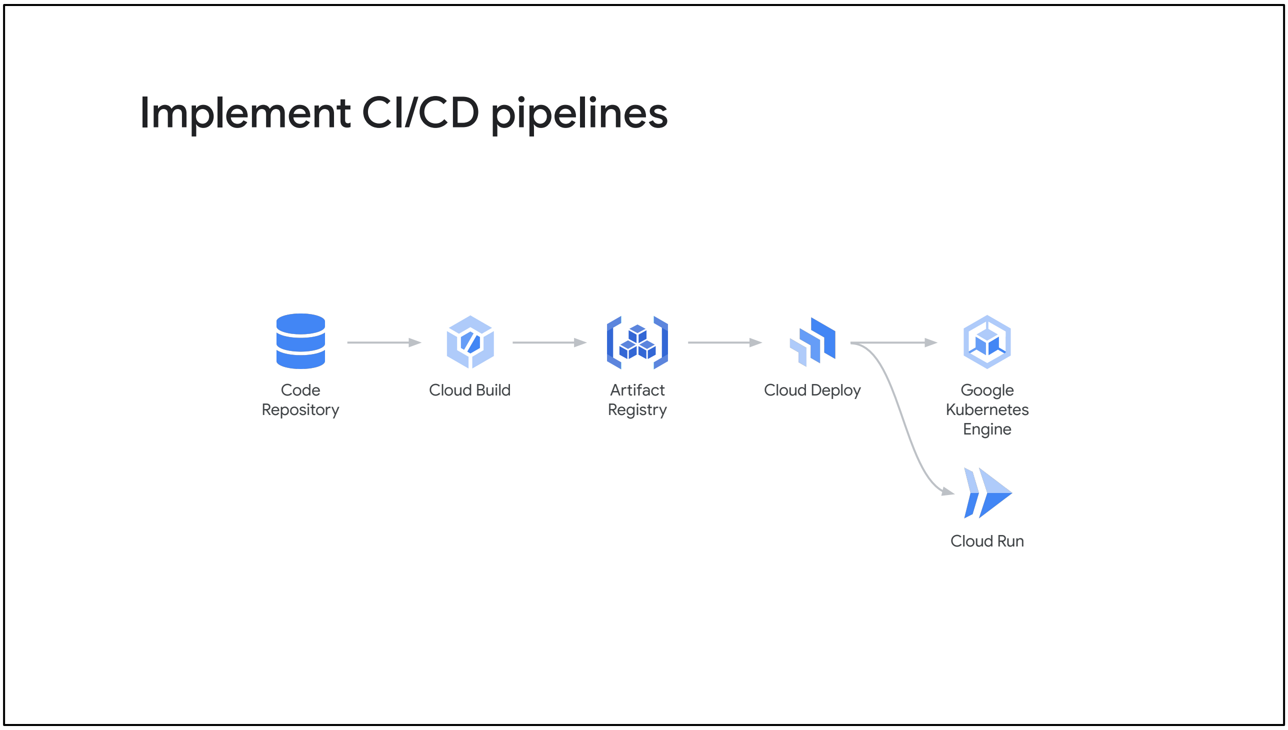
Implement a strong DevOps model with automation by using CI/CD pipelines. Automation helps you increase release velocity and reliability. With a robust CI/CD pipeline, you can test and roll out changes incrementally instead of making large releases with multiple changes. This approach enables you to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

CI is continuous integration, and CD is continuous delivery or continuous deployment.

In a continuous integration system, developers commit code changes made in their own feature branch into a shared branch in a code repository. When an update is made, the application build is triggered and automated testing confirms that the new update hasn't broken existing unit and integration tests.

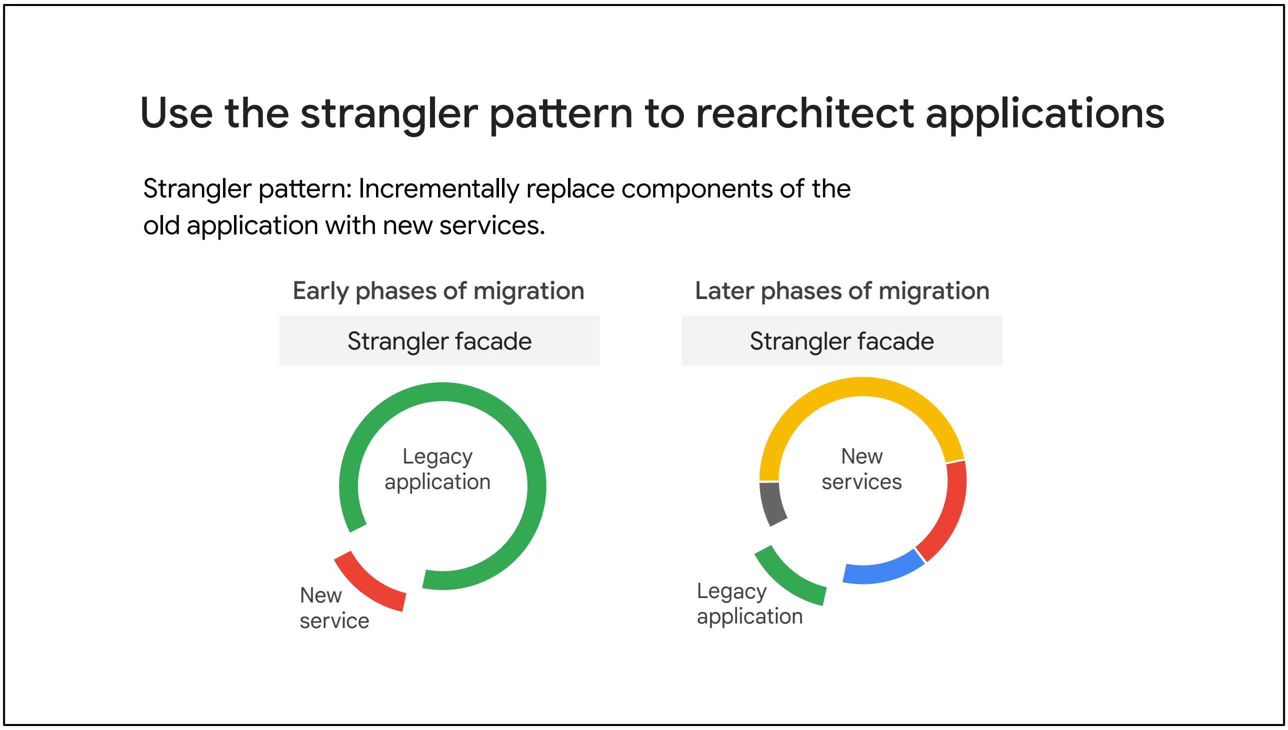
In a continuous delivery system, the code change validated during continuous integration is automatically stored in a code repository. This validated codebase is ready to be deployed to production by the operations team.

Continuous deployment takes this one step further by automatically deploying the validated changes to production. Only a failed test prevents the change from being deployed to production. Continuous deployment means that changes and fixes are available in production faster, and automated deployments reduce the work of your operations team. There is no manual process for validating the release to production, though, so your application, tests, and CI/CD pipeline must be well designed to prevent issues in production.



Implement CI/CD pipelines

When you create a CI/CD pipeline in Google Cloud, Cloud Build can detect repository commits, trigger a build, and run unit tests. The build system produces deployment artifacts that can be stored in Artifact Registry. Cloud Deploy can automatically trigger the deployment of builds to test environments or directly to production. 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 blue-green deployments or canary testing. These types of deployment strategies help ensure that unexpected issues with the new build do not affect most users.



Use the strangler pattern to rearchitect applications

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

Early phases of migration

Strangler facade

Legacy application

New service

Later phases of migration

Strangler facade

New services

Legacy application

Consider using the strangler pattern when rearchitecting or 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 allowing 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 enveloping the tree.

Google Cloud

As you create your own applications in the cloud, always consider these best practices. By following best practices of cloud development, you will set up your applications for success.