CI/CD Pipeline Design

Background

August would like to set up a custom CI/CD pipeline to deploy code for all client sites through staging and production environments. Code will be run as Docker containers, using a range of technologies (Node.js, PHP, Python etc.).

They do not already have an established CI/CD platform but are building out their workloads as runnable Docker containers with all code pushed to Github.

Planning

Key areas to understand before commencing solution design

Cloud vs On-Premise

When designing a CI/CD pipeline, one of the key decisions is determining whether to build the platform on a cloud provider (e.g. AWS/Azure/GCP) or on on-premise infrastructure (owned servers). Each option comes with its own advantages and disadvantages; and the right choice for the company can depend on what their current deployment environment looks like, the scale of operations, compliance needs, and willingness to manage infrastructure.

Criteria	Cloud Provider	On-Premise
Scalability	HIGHER Highly scalable, auto-scaling options	LOWER Limited by hardware; scaling requires new purchases and provisioning
Operational Overhead	LOWER Managed infrastructure	HIGHER You maintain hardware, updates, patches
Cost	Pay-as-you-go, can be expensive at scale	High upfront cost but predictable long-term cost
Customisation	LOWER Limited in managed services, high in self-managed setups	HIGHER Fully customisable
Security & Compliance	Built-in security services, but concerns with data residency in some industries	Full control over data and infrastructure, ideal for strict compliance needs
Disaster Recovery	SERVICE-MANAGED Cloud providers handle backups, DR, and failover	SELF-MANAGED You need to handle DR, backups, and replication manually
Global Reach	HIGHER Global infrastructure, edge locations for lower latency	LOWER Limited to the physical location of your servers

Upfront Capital Expense	LOWER Low to none (subscription model)	HIGHER High (hardware, networking, facilities)

One would also need to understand August's current environment when considering this choice. If they already have invested in on-premise servers to host their workloads, it may make more sense to build the CI/CD platform on top of it. However, a cloud environment is generally preferred due to the elastic scalability and managed services provided by the cloud provider.

Regarding choosing a cloud provider, the big 3 will have adequate capabilities to build a CI/CD pipeline. Considerations for choice would include cost/pricing model and the team's current familiarity/capability in using the service.

Centralised vs Distributed

This section is based on previous experience at MYOB, where we have shifted from distributed agents across ~350 AWS accounts to a centralised solution

Distributed agents are deployed across individual cloud accounts and offers lower latency for actions within the account, improved security through isolation and account specific customisation. However, this approach increases management complexity, potential resource duplication, operation overhead and build times as agents are more likely to be cold started.

Centralised agents are hosted in a single account which simplifies management and provides a consistent agent environment. Build times are generally faster as the agents are more likely to be "warm." Trade-offs include a broader access scope with cross-account roles required and lower fault tolerance as the agents may become a single point of failure.

Based on previous experience at MYOB, I would favour the centralised agents solution as it is much simpler to manage. However, August's needs for account-specific deployment configurations should be considered when making this decision.

Choosing a tool/vendor

CI/CD platform vendor

There are a number of vendors which provide CI/CD platform capabilities such as Buildkite, Jenkins, Github actions etc. as well as cloud native options (Azure DevOps Server, AWS CodePipeline). These tools likely provide similar capabilities, however for our use case it is important that the platform supports:

- Docker integration, as our workloads are Docker containers
- Github integration, as we want our builds to be triggered on code pushes
- Multi-stage builds, to reduce image size and enhance performance through Docker's caching.

Other considerations to take when choosing a vendor will include cost for business viability and ease of use to minimise friction in developer experience.

Tools present on the agent

It is important to also consider the tools that need to be installed on each of the agents to ensure they can perform the pipeline steps as part of the CI/CD process. Some of these tools can include:

- **Docker:** The main tool required to run our workloads. As the code is all containerised, the individual technologies (Node.js, php, python) do not need to be installed on our agents as they should be included in the Docker containers. However, Docker will be needed to run these containers.
- **Git:** When the pipeline gets triggered, one of the first steps for the agent would be to clone the repo in order to execute the code.
- **Tools required for deploying:** Depending on the deployment mechanism, this could be anything, but is specific to deploying the container. Some examples include: `awscli` for deploying to AWS, `kubectl` for Kubernetes etc.

- **Other command line tool examples:** `bash`, `yq`, `make` etc.

Align with the development team

Ensure you are aligned with the development team on their requirements and expectations as they are the main customer of the CI/CD platform. Ultimately, a CI/CD platform should be fast and easy to use to minimise disruptions in developer workflow. We should understand what their requirements are for each of the pipeline stages and build this into our design, some example requirements can include:

- **Development:** Fast feedback loops for developers, build times should be quick and should run unit tests, linting and static code analysis tools.
- **Staging:** Can deploy code to a pre-production environment with high environment parity to production. Validate the application behaviour before moving to production. Run a more thorough test suite such as integration and end-to-end testing.
- **Production:** Highly secure and controlled deployment with zero downtime required. Automated rollbacks in case of failure

Design Considerations

✓ Key considerations that should be factored into our design. Treat these more like guiding design principles that I would incorporate into the ideal design.

For the purpose of the exercise, these are written with a cloud-based environment in mind.

Built for failure

Any part of the system may break at any point, for any reason. High availability, observability and restorability should be built into the system as this platform is a business-critical operation, to ensure the developers can continue delivering.

Ideal state:

High availability: The system leverages elastic cloud computing capabilities and automatically scales accordingly to demand.

Observability: Adequate logging, metrics and traces are captured from the system. These help us monitor system performance and pinpoint the source of errors.

Restorability: Infrastructure is version controlled. Favour infrastructure as code (IaC) to define resources. Infrastructure is restorable to a n-1 version at any time, from scratch if needed.

Least privilege model

Any action that is allowed, is allowed for a reason. Any other action is denied.

Ideal state:

Permissions policies are based on actions whitelisting rather than blacklisting. Allowed actions should be documented along with justification to clarify the intent of inclusion.

Customer centred design

The user’s experience is at the core of the design and should be considered at each level. It should be the quickest and easiest solution for developers to deploy their code.

Ideal state:

The solution is frictionless to onboard, and is well documented.

The solution should not be seen as a black-box for users, but we should abstract the complexity away from the solution for those who are not interested in understanding further than how to use the platform.

Continuous improvement

Cloud solutions evolve rapidly, the design should be reviewed at regular intervals to ensure it stays relevant and adheres to current best practice.

Cost conscious

Consider the cost of the solution, strive to build solutions that are effective at solving the problem for the most *reasonable* cost.

Risks

⚠ Potential risks and how they might be mitigated

Security

Cloud Networking: Ensure the agent is not vulnerable to malicious attacks originating from the internet. We can mitigate this risk by protecting the agent by placing it in a private subnet in the cloud and restricting inbound and outbound traffic to/from the agent.

Credential Exposure: Hardcoding or improperly managing sensitive data such as API keys, secrets, and passwords in CI/CD pipelines can lead to leaks or unauthorised access. We can mitigate this risk by using a secrets manager (AWS Secrets Manager, HashiCorp Vault), passing sensitive data via environment variables or running security scanning tools in the pipeline (Gitleaks, Trufflehog) to notify the developer they have leaked the credential (although ideally we should check for leaked credentials at a pre-commit level).

Pipeline Authentication/Authorisation: Implement a strong authentication mechanism like OAuth or SSO to ensure users are authenticated before accessing the pipeline platform. Ensure only authorised users can access/act on certain pipelines, i.e. team-based or role-based access on pipelines. We should also audit access to pipelines regularly.

Container Security: As all workloads are run as Docker containers, we should strive to ensure they are as secure as possible. We can regularly scan images using tools like Trivy to monitor for vulnerabilities or outdated dependencies in our containers. We should consider running the containers in rootless mode to mitigate vulnerabilities. We can also take advantage of multi-stage builds to reduce the container’s attack surface.