Intuit Craft Project

Status	Done

Problem Statement

Developers in your BU lack a streamlined way to create the application resources they need when building new microservice web apps. This friction slows down development and product iteration. Developers need a simplified method for spinning up repositories, configuring build/release pipelines, and leveraging templates to kickstart development.

Directions

Kotlin Paved Road is an application that provides a streamlined, no-code wizard to create Kotlin applications from pre-defined template(s). The two functionalities that it provides are

- Create a Kotlin Library
- Create a Kotlin Web Service (based on spring ecosystem)

To provide the best experience to Kotlin developers, it is important that the resulting code should be ready to be used.

As part of a team helping the Kotlin developer community, your challenge is to think through the people, process, and technology decisions that will allow us to create and support this application with quality and speed for our Kotlin developers.

What is a paved road?

The concept of a "paved road" involves providing developers with established, standardized processes and tools to streamline their workflows. These pathways

are created by observing common practices and formalizing them into repeatable patterns. The primary goal is to offer a reliable and efficient route for development and deployment, reducing variability and increasing productivity by adhering to proven best practices. This approach helps teams avoid common pitfalls and focus on delivering value without needing to reinvent the wheel.

TL;DR: Developers can use established and standardized processes and tools, especially for boilerplates and deployments, without needing to reinvent the wheel.

What is the aim of this project?

The aim of this project is to build a product that allows the definition of various templates, such as those for a Spring Boot web app or a Kotlin-based library. The product should be designed to support the creation of any kind of language-agnostic template. The following assumptions outline the desired features and functionality:

Functional Requirements

- **Template Creation and Availability:** Allows developers to create standardized templates and make them available for organizational use.
- **App/Library Generation:** Enables developers to generate applications or libraries from these templates with minimal input.
- **Dynamic Configuration Forms:** Configuration forms for these templates should be dynamic and not require hard coding.
- **Dynamic Configuration Updates**: If a template config is updated, then the changes automatically carry forward to the paved-road project generation form.
- **Real-time Progress Tracking:** Developers can track the progress of their app/library creation in real time.
- **Service Linking and Cost Center Association:** Generated apps/libraries are saved as services linked to a cost center for the developer's business unit (BU).
- **Service Monitoring:** Services are accessible for monitoring purposes.

- **Deployment:** Developers can deploy their services from the paved-road UI/API.
- **Cost Observability:** The cost of the service is observable on the service page, based on the cost center and metadata linked to the service.
- **CI/CD Pipeline Configuration:** The paved road templates should also configure the CI/CD pipeline for deployment.
- **Modifiable Deployment Settings:** Developers can modify deployment settings (e.g., cluster size, memory) during the app/library creation process as well as later in their project page on the paved-road UI or through the paved-road API.

Non Functional Requirement

- **Scalability:** The generation of new apps/libraries from templates should scale easily to the entire organization.
- **Financial Reporting and Auditing:** Projects should be tied to the developer/team cost center for accurate financial reporting and auditing.
- **Performance and Efficiency:** Ensure that the system performs efficiently under expected loads and minimizes resource consumption.

Architecture

This document outlines the architecture and functionalities for a robust platform designed to streamline project generation and management. I have expanded upon the original scope to address a broader range of goals beyond simply generating projects from preconfigured templates. The key objectives and solutions incorporated into this project are as follows:

- **Template Creation** - Users have the capability to create and register templates for a diverse array of applications, irrespective of programming language or framework. This encompasses a wide variety of project types including REST APIs, data processing pipelines (e.g., Apache Spark, Apache Flink), event streaming services, frontend frameworks, and libraries.
- **Project Generation** - Users can utilize registered templates to generate and customize projects, ensuring flexibility and adaptability to specific project requirements.

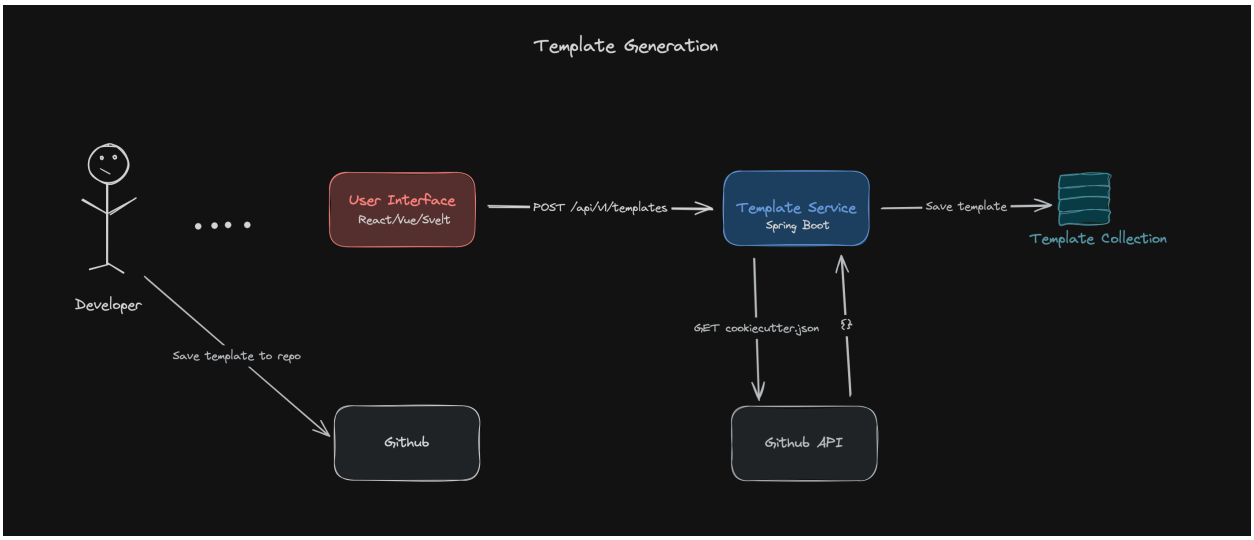
- **Project Metadata** - All generated projects are accessible via the paved-road portal, where they can be evaluated based on comprehensive metadata. This metadata includes the associated team and business unit (BU), cost center for streamlined financial auditing and reporting, a detailed audit of deployment/publishing activities, and the capability to deactivate or disable the project directly from the portal.
- **Cost Tracking** - Projects are linked to the user/team/BU that created them, with each project tagged to a specific cost center. This integration allows for detailed financial auditing and enables the tracking of project costs through cloud and internal API usage. The readily available cost information facilitates the configuration of alerts for cost optimization and potential savings.
- **CI/CD** - The platform supports comprehensive CI/CD processes by automatically creating GitHub repositories for projects, establishing Terraform-based deployment pipelines, and utilizing GitHub Actions for seamless deployment.
- **Deployment Customization** - Users can tailor their deployment strategies using the platform's UI along with customized configuration files within their projects.

Technology Stack

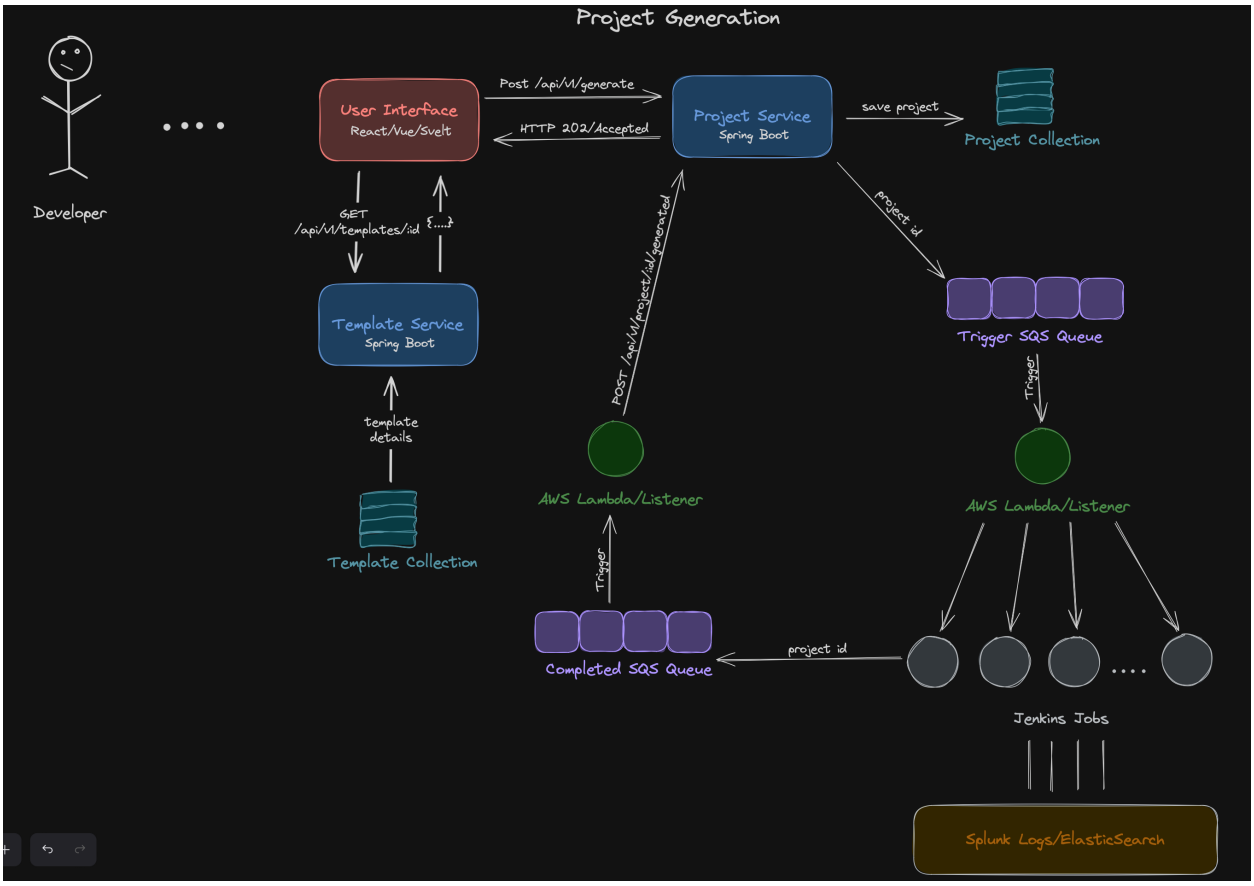
- **Template Generation**: Utilizes Cookiecutter, a tool that enables users to create and register their own custom templates through the paved-road UI or API.
- **Frontend Development**: Developed using React, TypeScript, and Tailwind CSS.
- **Backend Implementation**: The backend is powered by a Spring Boot REST API written in Kotlin.
- **Message Queueing**: AWS SQS (Simple Queue Service) for asynchronous messaging.
- **Message Queue Listener**: AWS Lambda functions.
- **Database** - I used mongoDB as this platform would require fairly simple and non-relational data models.
- **Cookiecutter Execution**: Jenkins or Github Actions.

- **CI/CD** - Integrates GitHub for source control, GitHub Actions for automated workflows, and Terraform for infrastructure as code to streamline continuous integration and deployment processes.

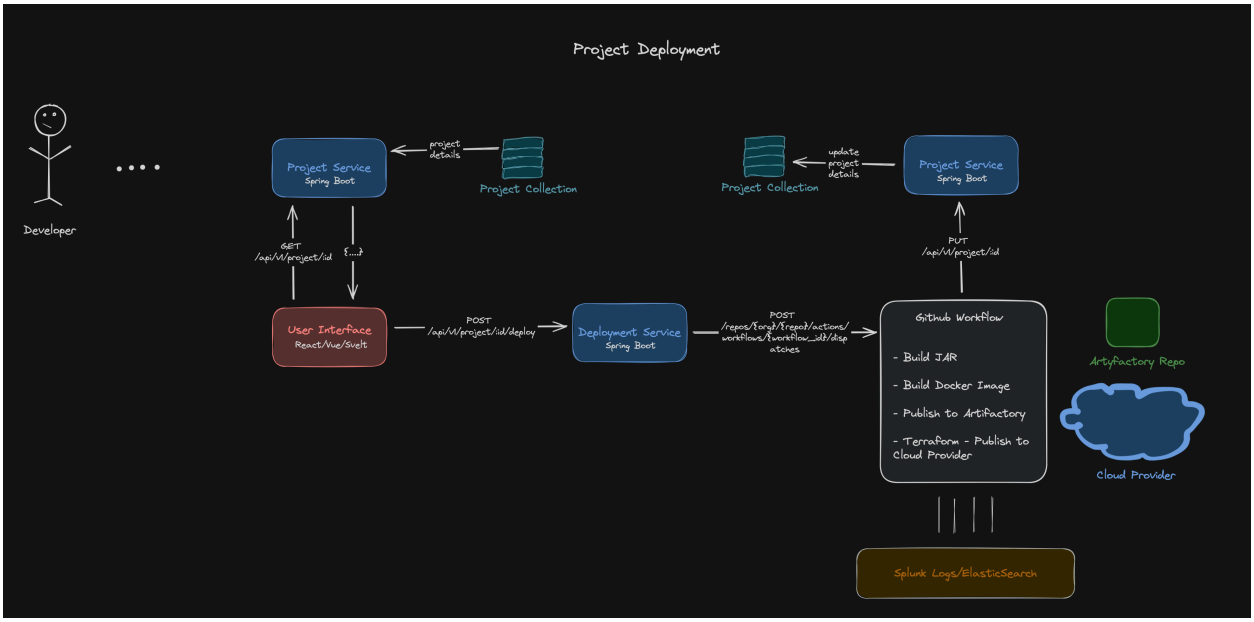
Template Creation Service



Project Generation Service



Deployment/Publishing Service



Data Models

Template

```
{
    id: uuid,
    createdAt: timestamp,
    createdBy: string,
    updatedAt: timestamp,
    updatedBy: string,
    name: string,
    tags: string[],
    description: string,
    config: {key: value}, // from cookiecutter.json
    projects: uuid[], // project.id
    health: boolean,
    reports: Report, // report.id based on user feedback
    ....

}
```

Project

```
{
    id: uuid,
    createdAt: timestamp,
  createdBy: string,
    updatedAt: timestamp,
    updatedBy: string,
    name: string,
    description: string,
    organization: string,
    author: User, // synced with ActiveDirectory
    tags: {
        application: string,
```

```
        product: string,
        organization: string,
        costCenter: string
        ....
    },
    template: uuid, // template.id
    repo: string,
    lastDeployment: timestamp || null,
    cloudProvider: CloudProvider, // cloud provider types
    artyfactoryImageLink: string,
    ....
}
```

Why Cookiecutter?

Cookiecutter is a versatile scaffolding tool designed to streamline the process of generating new projects from pre-configured templates. Users simply need to replace filenames and content in their templates with placeholders defined in the `cookiecutter.json` configuration file. When executed, Cookiecutter automatically replaces these placeholders with the appropriate values, creating a new project structure instantaneously.

Key Advantages:

- **Language Agnostic:** Works across various programming languages, offering flexibility regardless of the technology stack.
- **Simple Learning Curve:** Easy to understand and use, even for those new to project scaffolding.
- **Lightweight:** Minimal setup required, which makes it an excellent choice for quick project spin-ups.
- **Straightforward Configuration:** Utilizes simple JSON configuration files for template variables.
- **Active Community:** Supported by a broad and engaged community, ensuring continuous improvements and widespread use.

Alternatives:

- **Yeoman:** Predominantly used for JavaScript projects, Yeoman boasts a larger community but comes with a steeper learning curve. It integrates well with the paved-road framework, enhancing JavaScript project setups.
- **Maven Archetype:** Highly favored for its ease of use and widespread adoption within the Java ecosystem. While integrating seamlessly with the paved-road framework, creating new templates in Maven Archetype is notably more complex than in Cookiecutter and is restricted to Java-based projects.
- **Spring Initializer:** Offers a user-friendly UI and is optimal for setting up Java/Kotlin projects with Spring Boot. However, its use is confined to specific technology stacks, limiting its versatility compared to Cookiecutter.

Why Github Actions?

GitHub Actions and Spinnaker are both prominent CI/CD tools in modern software development. Spinnaker excels at configuring elaborate and complex pipelines and offers seamless monitoring capabilities with an intuitive UI that eases the learning process. However, GitHub Actions presents distinct advantages that make it a strong candidate for many CI/CD requirements.

I used github actions because of the following reasons -

- **Seamless GitHub Integration:** As a native feature of GitHub, Actions are seamlessly integrated, making it incredibly efficient to set up CI/CD pipelines within the same environment where the code is hosted.
- **Flexibility in Pipeline Creation:** GitHub Actions allows the creation of complex workflows that can handle multiple jobs, run in parallel, or sequentially depending on the needs of the project.
- **Extensive Tool Integrations:** GitHub Actions is highly compatible with a wide range of tools such as Terraform, Docker, and Jenkins, facilitating a robust and versatile CI/CD pipeline that can accommodate various needs.

Besides github actions, there are other tools which can be used in conjunction with Github Actions to ease the deployment pipeline creation process such as **Spinnaker**, **Argo Workflows**, **Actions Flow**, **CircleCI**, **Jenkins**, and **Octopus Deploy**.

Infrastructure as Code

We're using Terraform for the purposes of this demo but we are easily free to use other IaC tools as well. The benefits of terraform -

- Declarative
- Multi-cloud support
- Modular - allows you to reuse parts of the code
- Extensible - supports a wide range of providers and resources, and custom providers can be created to extend its functionality
- Large Community
- Progressive Deployment
- Automatic Rollback

Alternatives -

- Ansible
- Kubernetes and Helm Charts

Progressive Deployment

Github Actions and Terraform can used to progressively deploy our application thereby limiting the fallout of a bad deployment.

```
name: Progressive Deployment

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout code
      uses: actions/checkout@v2
```

```
    - name: Setup Terraform
      uses: hashicorp/setup-terraform@v2

    - name: Terraform Init
      run: terraform init

    - name: Terraform Plan Canary
      run: terraform plan -var="deployment_type=canary"

    - name: Terraform Apply Canary
      run: terraform apply -auto-approve -var="deployment_type=c

    - name: Health Check
      run: ./health_check_script.sh
      id: check

    - name: Terraform Apply Production
      if: steps.check.outputs.health == 'good'
      run: terraform apply -auto-approve -var="deployment_type=p
```

Why not Kubernetes?

I decided to write a few lines here since K8 is becoming very popular and inevitably crosses everyones mind as a solution when it comes things like paved-roads and federated CI/CD. While it is a very good and powerful solution, it’s not the one size fits all solution it sometimes gets made out to be.

- Steep learning curve
- Resource intensive
- Maintenance/Operationally Intensive
- Requires migration of existing infrastructure
- Useful for a large set of complex and highly distributed applications
- Hybrid and multi cloud setups

- Useful for advanced features like service mesh

Points of failure

1. **Web Portal/UI Failures**
    - Issues with user authentication and/or authorization **[SINGLE POINT OF FAILURE]**
    - Front end bugs which cause malfunction in any of the functionalities or disable them entirely
    - API endpoint failures
2. **Configuration Failures**
    - User defined configuration bugs - if a user defines an unsupported or malfunctioning configuration in their template then it can render the template unstable and unavailable
3. **Message Queue Failures**
    - Queue Overload - If the SQS queue becomes overloaded with too many messages, it could lead to delays or loss of messages **[SINGLE POINT OF FAILURE]**
    - Message Processing Failures - Issues with messages not being processed correctly or being lost **[SINGLE POINT OF FAILURE]**
4. Database Issues
    - Failure to connect with database **[SINGLE POINT OF FAILURE]**
    - Loss of data
5. Artyfactory Issues
    - Not being able to get/post images from/to arty due to problems with connecting to artyfactory repos
6. **AWS Lambda Failures**
    - Lambda functions may fail to execute due to code errors, configuration issues, or resource limits **[SINGLE POINT OF FAILURE]**
    - Could experience latency due to cold starts, affecting performance

7. GitHub Issues
    - Github being down is a single point of failure **[SINGLE POINT OF FAILURE]**
    - Some of the Github organizations/teams could be private leading to issues with project generation
    - Expired tokens being used to interact with Github can be a major single point of failure for the whole platform **[SINGLE POINT OF FAILURE]**
8. Project Generation Failures
    - Cookiecutter errors due to malformed configurations
9. Infrastructure Provisioning Failures
    - Terraform Errors - Issues with the Terraform scripts could lead to failed deployments
    - Cloud outages **[SINGLE POINT OF FAILURE]**
    - Exceeding cloud budget or resources
10. **Monitoring and Logging Failures**
    - Problems with the monitoring and logging services could prevent proper tracking of system performance and issues
11. Scalability and Performance Issues
    - Resource Exhaustion - Running out of CPU, memory, or other resources could lead to system failures
    - Scalability Limits - The platform might not scale effectively under heavy load, leading to performance degradation

Mitigation Strategies

1. Redundancy and Failover
    - Use multiple instances of critical services to ensure high availability
    - Implement failover mechanisms for key components
2. Monitoring and Alerting

- Use robust monitoring and alerting systems to detect and respond to issues promptly
- Implement automated health checks for critical services

3. Error Handling and Retries
    - Implement robust error handling and retry mechanisms for transient failures
    - Use dead-letter queues for failed messages that need manual intervention
4. Performance Optimization
    - Optimize code and configurations for performance
    - Use auto-scaling for handling variable loads
5. Security and Permissions
    - Ensure proper authentication and authorization mechanisms are in place
    - Regularly audit permissions and access controls
    - Have regular checks for expiring keys/tokens
6. Testing and Validation
    - Have validations in place to ensure incorrect configurations are easily found and reported - can use some GitHub bot to scan cookiecutter configurations with other template settings
    - Perform thorough testing, including unit, integration, and stress testing

Development Plan

1. **Stakeholder Identification and Engagement**
    a. **Identify Key Stakeholders**
        - **IT and Infrastructure Teams**: These teams know about the current infrastructure and technical/ business constraints and possibilities.
        - **Development Teams**: Primary users; their input crucial.
        - **Security Team**: This platform will hold a majority of the organizations technical infrastructure and products; we need their help to ensure it is

secure and safe.

- **Compliance and Audit Teams**: To ensure the solution meets all regulatory and compliance requirements.
- **Financial Team**: To understand financial constraints, estimate budgets and overheads.
- **Executive Team**: Crucial to get a go ahead for the project, tie this in with org strategy and priorities.

b. **Engage Stakeholders:**

- Conduct initial meetings to introduce the project, gather preliminary insights, and discuss potential impacts.
- Set up regular check-ins to keep stakeholders informed and involved throughout the project lifecycle.

2. **Requirement Gathering**

a. **Collect Detailed Requirements**

- **Technical Requirements:** Gather specific details about the existing technology stack, infrastructure, and any planned upgrades or changes.
- **Functional Requirements:** What the stakeholders expect the platform to achieve (e.g., faster deployment times, better scalability).
- **Non-functional Requirements:** Security, performance, scalability, reliability, and compliance.

b. **Methods for Requirement Gathering**

- **Interviews and Meetings:** One-on-one or group discussions with end-users and technical teams.
- **Surveys and Questionnaires:** To collect broad feedback from potential users across the organization.
- **Workshops:** Hands-on sessions to collaboratively define needs and solutions.

3. **Feasibility Study and Risk Analysis**

a. **Assess Feasibility**

- Evaluate technical and operational feasibility with the help of IT and infrastructure teams.
- Consider budget constraints and ROI analysis with finance stakeholders.

b. **Identify Risks**

- Technical risks such as integration challenges with existing systems, any migrations which will be required.
- Operational risks including changes in process that might disrupt current operations.
- Security and compliance risks.
- Document these risks and plan for mitigation.

4. **Planning and Roadmapping**

a. **Develop a Project Roadmap**

- **Phases and Milestones:** Break down the project into manageable phases, each with specific milestones and deliverables.
- **Timelines:** Establish realistic timelines for each phase, incorporating time for testing, feedback, and adjustments.
- **Resource Allocation:** Determine the resources (human, technological, financial) required for each phase.

b. **Tools and Techniques**

- Use JIRA or Trello or Notion etc to track progress.
- Implement Agile methodologies.

5. **Prototype and Pilot**

a. **Develop a Prototype:**

- Create a working model of the solution to demonstrate its feasibility and to refine requirements based on practical insights.

b. Pilot Program

- Select a pilot group from one of the development teams.
- Use the pilot to gather detailed feedback on the platform's performance and user experience, and make necessary adjustments.

6. **Implementation and Scaling**
    a. Gradual Rollout
        - After successful piloting, plan a gradual rollout to additional teams.
    b. Training and Support
        - Provide comprehensive training to all users.
        - Establish a support channels to address any issues as they arise during and after the deployment.
7. **Review and Iteration**
    a. Continuous Improvement
        - Regularly review the solution's performance against the set goals.
        - Gather user feedback continuously to make iterative improvements.
    b. Review Meetings
        - Regularly review the solution's performance against the set goals.
        - Gather user feedback continuously to make iterative improvements.
    c. Office Hours
        - Have an open and regular communication channel to help users onboard and address issues
8. **Rollback**
    a. **Disaster Planning**
        - Have a plan to rollback and shift infrastructure back to original solutions if things go sideways.

Test Plan

Objectives

- Ensure the platform functions as expected for all specified use cases.
- Validate that the system meets performance, security, and scalability requirements.
- Identify and resolve any defects or issues before production deployment.

Scope

- Functional testing of all features and user interactions.
- Non-functional testing including performance, scalability, security, and usability.
- Integration testing with external systems (GitHub, GCP, Terraform, etc.).

Testing Strategy

- Functional Testing
  - Areas to test
    - Web Portal/UI interactions.
    - User Authentication and Authorization.
    - Configuration Service.
    - Message Queue (SQS) functionality.
    - AWS Lambda functions.
    - Template Generation (Cookiecutter).
    - Project Repository interactions (GitHub).
    - Infrastructure Provisioning (Terraform, GCP).
    - Real-time monitoring and logging.
  - Test Cases
    - User login and authentication.
    - Template creation and availability.
    - App/library generation from templates.

        - Real-time progress tracking.
        - Service linking and cost center association.
        - Service monitoring and deployment.
        - Cost observability on service page.
        - CI/CD pipeline configuration.
        - Modifiable deployment settings.
- Non Functional Testing
    - Areas to test
        - Scalability of app/library generation process.
        - Performance under load.
        - Security of authentication and authorization.
        - Data integrity and privacy.
        - Usability of web portal and configuration forms.

Test Software

- Web portal, backend services, and all integrated systems.
- Test automation tools (Selenium - for imitating user interactions, JUnit - for unit and integration tests).
- Performance testing tools (JMeter - load testing tool).
- Security testing tools (ZAP - proxy which allows user to manipulate all traffic).

Metrics for Success

1. **Technical Metrics**
    - **Deployment Frequency**
        - Definition: The number of deployments over a specific time period.
        - What does it measure: Measures the agility and responsiveness of the CI/CD pipeline

- **Lead Time for Changes**
  - Definition**:** The time it takes for a commit to be made until it is successfully running in production.
  - What does it measure: Measures the efficiency of the deployment pipeline.
- **Change Failure Rate**
  - Definition**:** The percentage of deployments causing a failure in production.
  - Purpose**:** Indicates the stability, reliability, and robustness of the release process.
- **Mean Time to Recovery (MTTR)**
  - Definition**:** The average time taken to recover from a failure in the production environment.
  - Purpose**:** Measures the resilience and effectiveness of the incident response and rollbacks.
- **Availability/Uptime**
  - Definition**:** The percentage of time the system is functional and working as expected.
  - Purpose**:** Tracks the reliability and operational performance of the deployed infrastructure.

2. **User Adoption Metrics**
   - Active Users
   - User Satisfaction Score
   - Support Questions/Engagements - This in itself should not be directly used as a measure for adoption but when compared with the number of active users, can tell us how difficult a time users are having onboarding to the platform.
3. Business Impact Metrics

- Return on Investment (ROI) - Calculated by comparing the savings/gain against the cost of the project.
- Productivity Metrics
    - Definition**:** Changes in productivity levels, measured through output per team or individual before and after implementation.
    - Purpose**:** Assesses how the project has impacted the overall productivity of the teams using it.
- Cost Savings
    - Definition**:** Reduction in costs associated with project operations, including reduced manpower due to automation, lower maintenance costs, lower compute cost, and decreased downtime.
    - Purpose**:** Quantifies the cost efficiency gained through the project.

4. Quality Metrics
    - Lower Bug Rates
    - Higher Code Coverage
5. Compliance and Security Metrics
    - Improvement of the number of compliance audits passed
    - Reduction in security incidents

SLAs

- **Availability (Uptime) SLA**: This SLA specifies the percentage of time the service should be operational and accessible without unscheduled downtime. Eg. 99.999 or 99.9999
- **Response Time**: This SLA sets the expected time for the system to respond to a user's request, typically measured in seconds or milliseconds.
- **Throughput**: Measures the number of transactions or tasks the system can handle within a certain period.
- **Incident Response Time**: The amount of time it takes for the support team to address and resolve a reported issue.

- **Problem Resolution Time**: The expected time within which an issue should be resolved after it has been reported.
- **Security Incident Response Time**: The timeframe within which the provider must respond to and address security incidents.
- **Recovery Time Objective (RTO)**: The maximum tolerable duration of time that a service process can be down after a disaster before the stated business objectives are adversely affected.
- **Recovery Point Objective (RPO)**: The maximum tolerable period in which data might be lost due to a major incident.
- **Change Request Turnaround**: An SLA for the time taken to implement a change request from a user or to deploy a new release, ensuring minimal disruption to the live environment.

Product

Identifying and prioritizing customer needs

- Platform infrastructure teams who would have access to monitor internal as well as cloud based resources the organization is using can conduct an audit to identify patterns in usage, which are optimal patterns and which ones can be improved, which ones are needlessly expensive, and which ones cause development lag.
- Additionally, feedback can be gathered from developers and managers across the organization to identify key paint points
- Based on this process of information gathering we divide our priorities between `Must Have, Should have, Could have, Wont have` i.e. the MoSCoW prioritization framework.

Key Outcomes and Metrics

- Reduction in average product set up time
- Reduction in average deployment time
- Reduction in average number of daily deployment failures

- Increased developer productivity measured by reduced project completion times and increase in number of stories completed per developer
- Reduction in reported bugs
- Reduction in average time to fix a bug
- Improvement in code quality measured by better and standardized logging, better monitoring and telemetry
- Standardized coding practices

Validate if the solution matches user problems

- Continuous monitoring of usage metrics and user behaviour
- Open and regular feedback channels

Technology

Critical aspects of the architecture

- Scalability to handle multiple concurrent project creations
- Allowing a diverse variety of user generated templates
- Monitoring the health of templates by looping in feedback from service deployment and user reported issues
- Integration with a variety of CI/CD systems like Jenkins, Github Actions, Spinnaker etc

Domain Model

- Template - represents a user generated template for a diverse range of softwares like a spring boot web app, library, python web app, streaming app etc
- Project - a project generated from a template
- Developer - user who generates a project
- Team - team which owns a template and and/or a project
- Business Unit - tied to cost center

- Repository - where the project is stored
- Build Pipeline - CI/CD configuration

Dependency Management

Dependency management hell is real and has been faced by almost every developer. One of the key problems a template based project generation framework can solve is to minimize a situation where a developer has to burn in dependency hell.

- Templates generated by a developer/team will have a standardized working list of dependencies which would have been tested prior to being added to the paved-road platform.
- Looping in feedback about template health from users and by checking problems related to the template in terms of deployment or bugs will ensure keeping the users aware of which templates are healthy and which are not.
- Further, we can integrate dependency management tools like Renovate and Dependabot which can scan the repositories and make federated updates to them based on security issues, bugs, and/or dependency management fixes

Product Development

Success Metrics

- Number of projects set up using the platform
- Number of templates
- Number of healthy vs unhealthy templates
- Developer satisfaction scores
- Measured cost savings compared to existing or previous system
- Measured improvement developer productivity scores

Stakeholders

Ensuring regular feedback and an open communication channel with our stakeholders will be essential. Further, we'll need to have a detailed

documentation and a communication channel to easily onboard developers and answer support questions.

- Development teams - primary impact
- IT operations - secondary impact
- Senior Management

Delivery Sequence

- Stage #1 - MVP with a limited templates; a UI and an api; allows template based project generation
- Stage #2 - Allow custom templates
- Stage #3 - Integration with CI/CD
- Stage #4 - Advanced customization options
- Stage #5 - Feedback loop for project monitoring
- Stage #6 - Multiple cloud deployments

Additional thoughts

Depending on the scope of the project it can be extended to include -

- Service mesh - allowing side cars to deployments can solve a variety of issues related to auth/authz, logging, monitoring and improve developer productivity and best practices even more
- Federated deployment - this can have fast resolutions to org wide issues like major bug fixes, security breaches etc