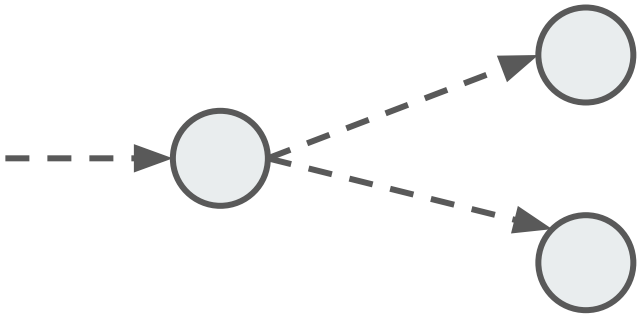
Pragmatic DevOps

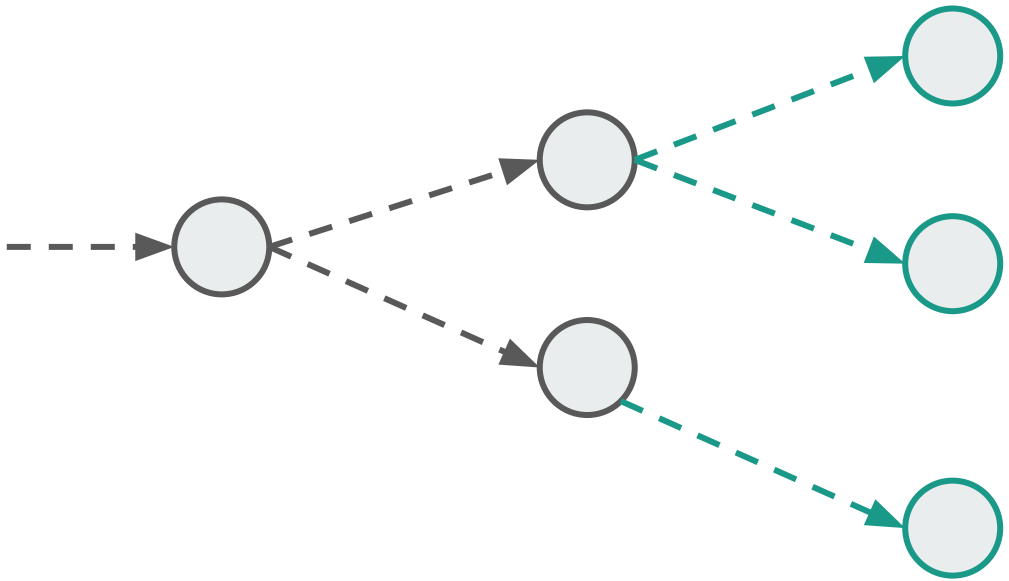
Starting up in 2013

Simple Architecture



Late 2015

Moving to microservices



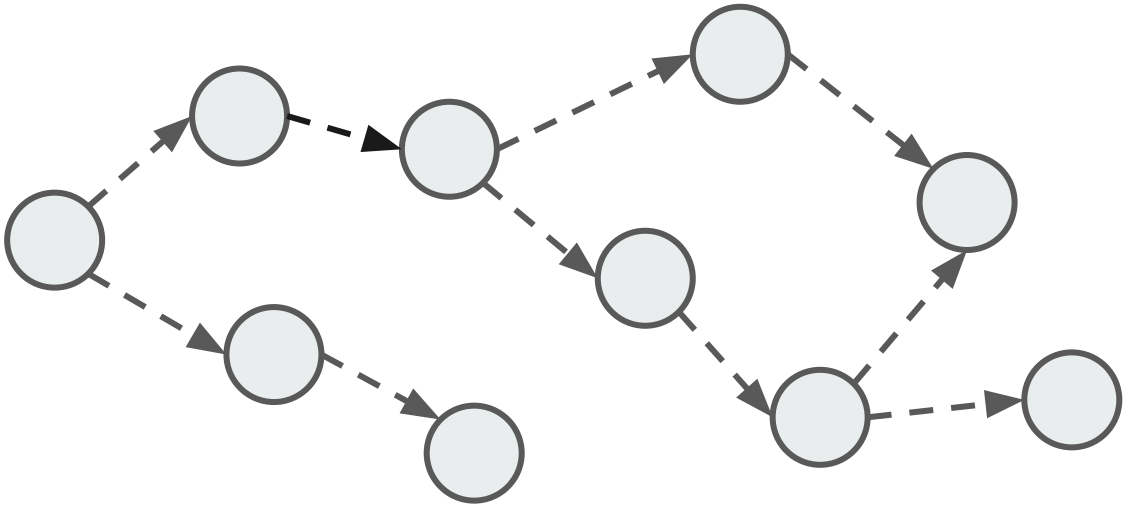
Early 2016

You build it, you run it

Early 2018

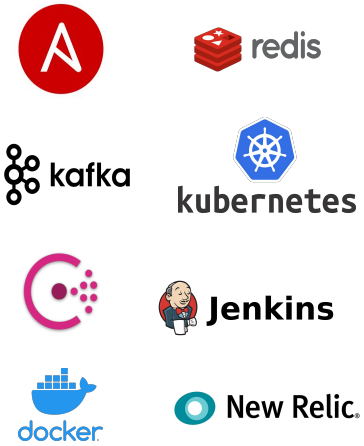
Illusion of Agility

Proliferation of Microservices



Proliferation of Tech Stack

INFRASTRUCTURE



PROGRAMMING LANGUAGES

DATABASES

What's wrong?

- Not able to improve test coverage
- Developer experience not improving
- State of quality not improving

Some progress

- Realistic goals
- Team level localized goals to attack localized problems
- Focus on “critical” services
- Some goals met but no progress on writing tests

Lesson 1:

There is no such thing as best practice that you must follow.

Lesson 2:

Engineering practices get adopted faster when there is a clear execution plan with outcomes attached.

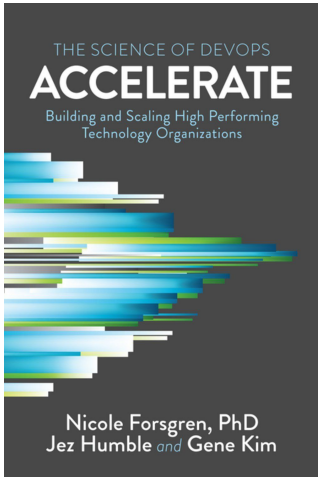
Lesson 3:

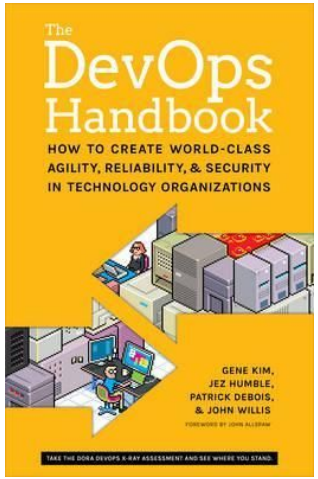
Reality will push you to prioritize if you don't. This applies to adopting engineering practices as well.

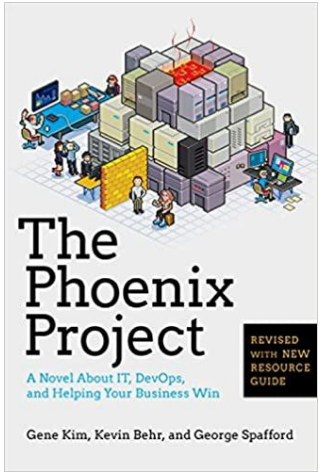
Lesson 4:

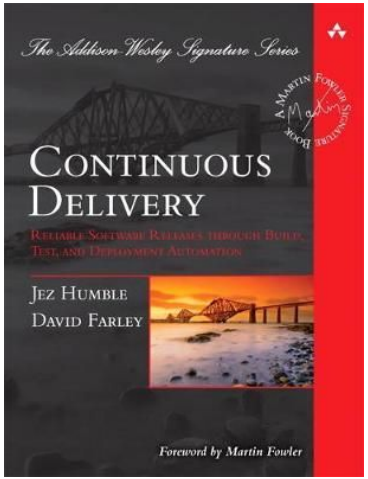
Every team is at a different journey. Different practices are needed for different teams and systems.

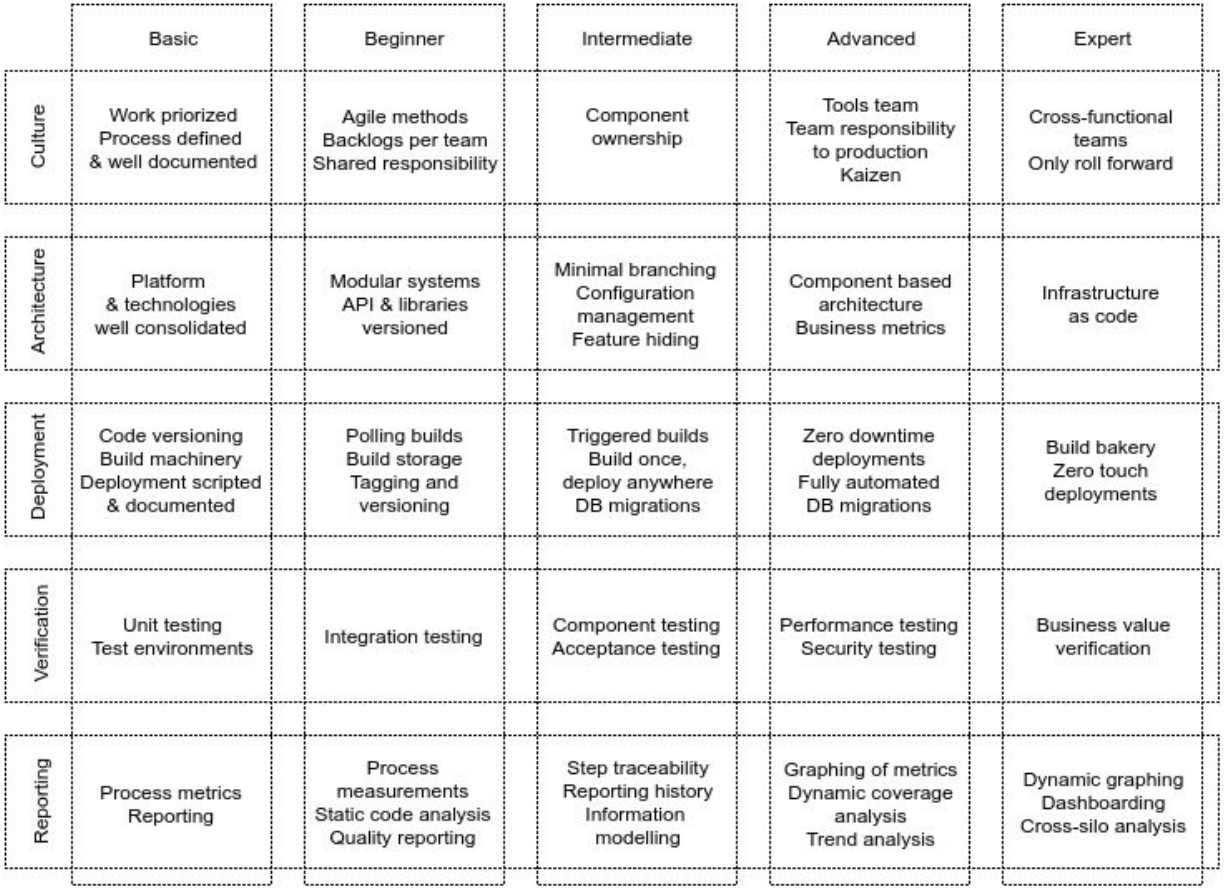
Wander & Learn











	Basic	Beginner	Intermediate	Advanced	Expert
Culture	Work priorized Process defined & well documented	Agile methods Backlogs per team Shared responsibility	Component ownership	Tools team Team responsibility to production Kaizen	Cross-functional teams Only roll forward
Architecture	Platform & technologies well consolidated	Modular systems API & libraries versioned	Minimal branching Configuration management Feature hiding	Component based architecture Business metrics	Infrastructure as code
Deployment	Code versioning Build machinery Deployment scripted & documented	Polling builds Build storage Tagging and versioning	Triggered builds Build once, deploy anywhere DB migrations	Zero downtime deployments Fully automated DB migrations	Build bakery Zero touch deployments
Verification	Unit testing Test environments	Integration testing	Component testing Acceptance testing	Performance testing Security testing	Business value verification
Reporting	Process metrics Reporting	Process measurements Static code analysis Quality reporting	Step traceability Reporting history Information modelling	Graphing of metrics Dynamic coverage analysis Trend analysis	Dynamic graphing Dashboarding Cross-silo analysis

Continuous Delivery Maturity Model (Continuous Delivery - Dave Farley, Jez Humble)

Pillar	Areas	Level 1 - Repeatable Process documented and partially automated Example: invoicing, service-inventory		Level 2 - Consistent Automated processes applied across whole application lifecycle Example: drop_shippinng, segment-api		Level 3 - Quantitatvely Managed Process measured and controlled Example: collections, cash-recon (for compliance)		Level 4 - Optimizing Focus on process improvement continuously Example: kong, iris, presentation-layer, bifrost	
		Expectation	**Support at Grofers**	**Expectation**	**Support at Grofers**	**Expectation**	**Support at Grofers**	**Expectation**	**Support at Grofers**
Ease of Development	**Development Infrastructure**	- Can be manually prepared. New team members find it impossible to get up and running by themselves. Always **need another person on the team to hand hold**.	- README with the description and basic information of the service must be present. Incomplete README is okay but not having a README is not okay. Also refer C24	- Can be **setup in maximum a couple of hours** using repeatable scripts or instructions as documented in the README. **Doesn't need handholding by another person on the team.**	- Steps for installing dependencies, setup and running tests are documented in project's README - Depedency installation and application runtime is dockerized - Config file templates exist in the project repo and working config can be pulled from an already running environment Also refer E24	- Can be **setup in minutes** using containerized infrastructure. Should resemble production. If needed, develop against an integrated environment.	- Single command to setup an integrated debugging environment for locally running application via **telepresence** or **skaffold** - Configuration should be automatically generated from Consul instead of being manually entered using the same config-templates as for other envs - **Kustomize** is used for overriding any dev/local specific config - Helm is used with Skaffold for setting up open-source services like postgres, redis, mongo, etc. - Dev friendly Dockerfiles. Can we save state of docker work somehow? Can we pipe commands from terminal to docker container so that bash history is not lost? .dockerignore to support thin docker images locally.	- **Fully integrated environment for local development is not needed**. Development can reliably happen against **stubs/mocks/contracts (also see contract testing in the "Testing" section)**.	- Services and dependecies can be setup locally with one step - Critical depedencies should be mocked using pact/mountebank/etc so that service can indepedently come up. Apparantly API gateways can also be used for mocking.
	Configuration Management	- Secrets are stored in an external secret storage system and not committed in SCM - Configuration files are templatized and config values are version controlled per environment - Config is version controlled with the application code - Every config change needs a manually triggered deployment.	- Secrets are stored in Vault - Applications get secrets using HVAC, consul-template - Configuration is stored in Ansible repos or Consul. - Applications get configuration using consul library or consul-template. - KVs in Consul are version controlled using ConsulConfigMap - How do we streamline config deployment? It's broken and differently done everywhere right now. Explore consul-k8s	- Configuration management across multiple envs (prod, stage, CI, dev) is easy	- Use Kustomize with repositories for manging environment specific configuration and manifests (including KVs in Consul using ConsulConfigMap). Uses Ansible to achieve the same thing for workloads on EC2.	- Every system config change triggers a graceful application reload to pick up new configuration change instead of a full deployment. - Every application config change can reload the cofnig context of the application (i.e. doesn't require an application process restart).	- Applications gracefully handle config updates by auto-reloading applications using consul-template. - Application config is managed in consul, differently as compared to how we manage system configuration. Applications can defined TTL for config expiry according to their business requirements so that not every app config change requires the service to be restart.	- Uses dynamically generated secrets with attached lease for periodic secret rotation. - Applications use IAM roles instead of secrets for IAM users	- Secrets with dynamic credentails are requested using consul-template by applications from Vault. Secrets will have leases attached. Leases get auto-expired. Should be done wherever possible. - Applications gracefully handle expirty of secrets with. leases by auto-reloading applications using consul-template. - kube2iam or an alternative for this
			- Use declarative Jenkins pipeline (i.e. Jenkinsfile) to setup CI pipeline. - Docker images with standardized tags are created and pushed for git pushes for server apps. Refer this for current standard - https://grofers.atlassian.net/wiki/spaces/RELENG/pages/1501593601/Best+Practices#Docker-tagging	- Automated build and test cycle (with more types of tests) every time a change is committed (on every push to a pull request). Dependencies managed through	- The CI pipeline should follow this high level standard -				

Microservices Maturity Model at Grofers

Pillars & Areas

Ease of Development	Releasability	Operations & Resilience	Security & Compliance	Organization, Culture & Processes
Development Infrastructure	Configuration Management Build and CI Environments and Deployments Release management and compliance Testing Data management	Service Resilience Performance Testing & Capacity Planning Synthetic Monitoring Monitoring Logging Tracing Alerting Incident Management	Application Security Audit Logging Risk Management Change Management Inter-Service Communication Internal & external audits	Cost Reporting Documentation Tech Debt Management Inner-Source

Pillars & Areas - for an e-commerce biz?

Ease of Development	Releasability	Operations & Resilience	Security & Compliance	Organization, Culture & Processes
Development Infrastructure	Configuration Management Build and CI Environments and Deployments Release management and compliance Testing Data management	**Service Resilience** Performance Testing & Capacity Planning Synthetic Monitoring **Monitoring** Logging Tracing Alerting Incident Management	Application Security Audit Logging Risk Management Change Management Inter-Service Communication Internal & external audits	Cost Reporting Documentation Tech Debt Management Inner-Source

Pillars & Areas - for a fintech biz?

Ease of Development	**Releasability**	Operations & Resilience	**Security & Compliance**	Organization, Culture & Processes
Development Infrastructure	Configuration Management Build and CI Environments and Deployments Release management and compliance **Testing** Data management	Service Resilience Performance Testing & Capacity Planning Synthetic Monitoring Monitoring Logging Tracing Alerting Incident Management	Application Security Audit Logging Risk Management Change Management Inter-Service Communication **Internal & external audits**	Cost Reporting Documentation Tech Debt Management Inner-Source

Pillars & Areas - for a SaaS biz?

Ease of Development	Releasability	Operations & Resilience	Security & Compliance	Organization, Culture & Processes
Development Infrastructure	Configuration Management Build and CI Environments and Deployments Release management and compliance Testing Data management	**Service Resilience Performance Testing & Capacity Planning** Synthetic Monitoring **Monitoring** Logging Tracing Alerting Incident Management	Application Security Audit Logging Risk Management Change Management Inter-Service Communication Internal & external audits	Cost Reporting Documentation Tech Debt Management Inner-Source

Prescriptive

Pillar	Areas	Level 1 - Repeatable Process documented and partially automated Example: invoicing, service-inventory		Level 2 - Consistent Automated processes applied across whole application lifecycle Example: drop_shippinng, segment-api	
	Examples	Expectation	Support at Grofers	Expectation	Support at Grofers
	Synthetic Monitoring	- Synthetic monitoring is setup to poll the healthcheck endpoint fo the servie and alert (and escalate) the team. Refer to C16	- Platform to easily setup synthetic monitoring needs to be built. Ideally should run like Lambda. Perhaps integrate with Opsgenie's heartbeat. - Default healthcheck at service level is implemented as part of the platform for stateless applications. - Stateful applications must implement their health checks. Use the base synethetic monitoring platform to schedule healthchecks and alert on failure. Refer to D16	- Synthetic monitoring is used in production with **some alerting**	- Services implement a well-defined **smoke suite (P1 cases)** that can be run in all the enviroments and periodically in production using Jenkins. Can potentially use Argo Workflows here (would be application in all the levels). - Need a guideline for how synthetic monitoring should be setup. For example, how to implement, automate, schedule, alert, frequency of execution, etc. Alerts should use the same incident management process (i.e. OpsGenie + some kind of status page). - Should be done in collaboration with testing team and product managers.
	Monitoring	Service level metrics (RPM. Latency) and system level metrics (Disk, CPU, Memory, etc.) are available. Use out-of-the-box monitoring (home grown or available as a service) wherever possible.	- Prometheus and Grafana for visualization of metrics. - Uses out-of-the-box monitoring (ELB metrics, ASG metrics, infra metrics from EC2 and Prometheus, application specific prometheus exporters, etc. - choose whatever is applicable or works). - Uses Sentry for exception tracking. - Use Legend for dashboarding if possible.	- Endpoint level metrics are always available (e.g. latency per endpoint). - Custom metrics are added on need basis.	- Uses application specific Prometheus exporters to ship common metrics at endpoint level - If framework specific exporters are not available, implements custom exporters for the runtime. - Custom metrics are exposed using Prometheus (InfluxDB is deprecated).
	Logging	All out-of-the-box logs are collected. At least incoming requests for HTTP services are logged (e.g. nginx logs, gunicorn logs, etc.).	- Uses Loki for logging (logs to stdout). Loki support for EC2 needs to be added. Devs will hav eto use watch_logs ansible module to enable log tailing. - Open-source frameworks / systems should have logging properly enabled and configured	Application specific custom logs are there. May or may not follow consistency.	- Use standard library of your application's ecosystem.
	Tracing				

Prescriptive

Pillar	Areas / Examples	Level 1 - Repeatable Process documented and partially automated Example: invoicing, service-inventory Expectation	Support at Grofers	Level 2 - Consistent Automated processes applied across whole application lifecycle Example: drop_shippinng, segment-api Expectation	Support at Grofers
	Synthetic Monitoring	- Synthetic monitoring is setup to poll the healthcheck endpoint fo the servie and alert (and escalate) the team. Refer to C16	- Platform to easily setup synthetic monitoring needs to be built. Ideally should run like Lambda. Perhaps integrate with Opsgenie's heartbeat. - Default healthcheck at service level is implemented as part of the platform for stateless applications. - Stateful applications must implement their health checks. Use the base synethetic monitoring platform to schedule healthchecks and alert on failure. Refer to D16	- Synthetic monitoring is used in production with **some alerting**	- Services implement a well-defined **smoke suite (P1 cases)** that can be run in all the enviroments and periodically in production using Jenkins. Can potentially use Argo Workflows here (would be application in all the levels). - Need a guideline for how synthetic monitoring should be setup. For example, how to implement, automate, schedule, alert, frequency of execution, etc. Alerts should use the same incident management process (i.e. OpsGenie + some kind of status page). - Should be done in collaboration with testing team and product managers.
	Monitoring	Service level metrics (RPM. Latency) and system level metrics (Disk, CPU, Memory, etc.) are available. Use out-of-the-box monitoring (home grown or available as a service) wherever possible.	- Prometheus and Grafana for visualization of metrics. - Uses out-of-the-box monitoring (ELB metrics, ASG metrics, infra metrics from EC2 and Prometheus, application specific prometheus exporters, etc. - choose whatever is applicable or works). - Uses Sentry for exception tracking. - Use Legend for dashboarding if possible.	- Endpoint level metrics are always available (e.g. latency per endpoint). - Custom metrics are added on need basis.	- Uses application specific Prometheus exporters to ship common metrics at endpoint level - If framework specific exporters are not available, implements custom exporters for the runtime. - Custom metrics are exposed using Prometheus (InfluxDB is deprecated).
	Logging	All out-of-the-box logs are collected. At least incoming requests for HTTP services are logged (e.g. nginx logs, gunicorn logs, etc.).	- Uses Loki for logging (logs to stdout). Loki support for EC2 needs to be added. Devs will hav eto use watch_logs ansible module to enable log tailing. - Open-source frameworks / systems should have logging properly enabled and configured	Application specific custom logs are there. May or may not follow consistency.	- Use standard library of your application's ecosystem.
	Tracing				

Risk Driven

~~Not Aspirational~~

- Level of microservices decided systematically by using parameters like frequency of changes, active collaborators, age of codebase, is it in the critical path to serving the customer, etc.
- Recalculate levels periodically to assess change in landscape.
- Use this information to prioritize technology investments like improving practices or changing architecture

Pillar	Areas	Level 1 - Repeatable Process documented and partially automated Example: invoicing, service-inventory		Level 2 - Consistent Automated processes applied across whole application lifecycle Example: drop_shippinng, segment-api		Level 3 - Quantitatvely Managed Process measured and controlled Example: collections, cash-recon (for compliance)		Level 4 - Optimizing Focus on process improvement continuously Example: kong, iris, presentation-layer, bifrost	
	Examples	Expectation	Support at Grofers	Expectation	Support at Grofers	Expectation	Support at Grofers	Expectation	Support at Grofers
Ease of Development	**Development Infrastructure**	- Can be manually prepared. New team members find it impossible to get up and running by themselves. Always **need another person on the team to hand hold.**	- README with the description and basic information of the service must be present. Incomplete README is okay but not having a README is not okay. Also refer C24	- Can be **setup in maximum a couple of hours** using repeatable scripts or instructions as documented in the README. **Doesn't need handholding by another person on the team.**	- Steps for installing dependencies, setup and running tests are documented in project's README - Depedency installation and application runtime is dockerized - Config file templates exist in the project repo and working config can be pulled from an already running environment Also refer E24	- Can be **setup in minutes** using containerized infrastructure. Should resemble production. If needed, develop against an integrated environment.	- Single command to setup an integrated debugging environment for locally running application via **telepresence or skaffold** - Configuration should be automatically generated from Consul instead of being manually entered using the same config-templates as for other envs - **Kustomize** is used for overriding any dev/local specific config - [illegible] - Dev friendly Dockerfiles. Can we save state of docker work somehow? Can we pipe commands from terminal to docker container so that bash history is not lost? .dockerignore to support thin docker images locally.	- **Fully integrated environment for local development is not needed**. Development can reliably happen against **stubs/mocks/contracts (also see contract testing in the "Testing" section).**	- Services and dependecies can be setup locally with one step - Critical depedencies should be mocked using pact/mountebank/etc so that service can indepedently come up. Apparantly API gateways can also be used for mocking.
Releasability	**Configuration Management**	- Secrets are stored in an external secret storage system and not committed in SCM - Configuration files are templatized and config values are version controlled per environment - Config is version controlled with the application code - Every config change needs a manually triggered deployment.	- Secrets are stored in Vault - Applications get secrets using HVAC, consul-template - Configuration is stored in Ansible repos or Consul. - Applications get configuration using consul library or consul-template. - KVs in Consul are version controlled using ConsulConfigMap - How do we streamline config deployment? It's broken and differently done everywhere right now. Explore consul-k8s	- Configuration management across multiple envs (prod, stage, CI, dev) is easy	- Use Kustomize with repositories for manging environment specific configuration and manifests (including KVs in Consul using ConsulConfigMap). Uses Ansible to achieve the same thing for workloads on EC2.	- Every system config change triggers a graceful application reload to pick up new configuration change instead of a full deployment. - Every application config change can reload the cofnig context of the application (i.e. doesn't require an application process restart).	- Applications gracefully handle config updates by auto-reloading applications using consul-template. - Application config is managed in consul, differently as compared to how we manage system configuration. Applications can defined TTL for config expiry according to their business requirements so that not every app config change requires the service to be restart.	- Uses dynamically generated secrets with attached lease for periodic secret rotation. - Applications use IAM roles instead of secrets for IAM users	- Secrets with dynamic credentails are requested using consul-template by applications from Vault. Secrets will have leases attached. Leases get auto-expired. Should be done wherever possible. - Applications gracefully handle expirty of secrets with. leases by auto-reloading applications using consul-template. - kube2iam or an altemative for this
	Build and CI	Automated build and testing. Any build can be re-created from source control using automated process on-demand. Test/build environment could be shared, making it hard to run tests in parallel.	- Use declarative Jenkins pipeline (i.e. Jenkinsfile) to setup CI pipeline. - Docker images with standardized tags are created and pushed for git pushes for server apps. Refer this for current standard - https://grofers.atlassian.net/wiki/spaces/RELENG/pages/1501593601/Best+Practices#Docker-tagging - Docker images are built and pushed. Can be implemented using CI Conductor. Language ecosystem specific support can be used (for example, standard process for Python, Java, etc.). Artifact repositories can be used to persist builds. - What about frontend clients like andoird app and web?	- Automated build and test cycle (with more types of tests) every time a change is committed (on every push to a pull request). Dependencies managed through code. Re-use of scripts and tools. Tests are run in isolated environments for every PR, making it possible to run tests for changes in parallel. - Automated tests / checks are required to pass before a pull request can be merged. Also refer to E7 and E8	- The CI pipeline should follow this high level standard - https://github.com/grofers/gr-project-template/blob/master/docs/ci-conductor.md#the-pipeline-structure - CI pipleines should create new environment for every PR. Use mft on Kubernetes. For services on EC2, this is not possible and hence these services should migrate to Kubernetes.	Build metrics gathered, made visible and acted on. Builds are not left broken	CI pipeline should emit following metrics - - CI health: - Stability rate of env setup - Speed of orchestration of CI environment (Monitoring needs to be built, more metrics will be added. See this example)	Teams regularly meet to discuss integration problems and resolve them with automation, faster and better visibility	- CI pipeline/practice health reports are reviewed every sprint within the team during retros. Reports should be readily available to every team.
	Environments and Deployments	Automated deployments to some environments (prod and stage,	Automated deployments to some environments (prod and stage,	Fully automated, self service push-button process for	- Use common Jenkinsfile and Kubernetes manifests with	Orchestrated deployments managed. Release and rollback	- Deployment is triggered automatically on creating a	Deployments, promotion and rollback on the basis of SLOs and	- Rollout, promotion, rollbacks done using something like

Platform Thinking Baked In

Pillar	Areas	Level 1 - Repeatable Process documented and partially automated Example: invoicing, service-inventory		Level 2 - Consistent Automated processes applied across whole application lifecycle Example: drop_shippinng, segment-api		Level 3 - Quantitatvely Managed Process measured and controlled Example: collections, cash-recon (for compliance)		Level 4 - Optimizing Focus on process improvement continuously Example: kong, iris, presentation-layer, bifrost	
	Examples	Expectation	Support at Grofers	Expectation	Support at Grofers	Expectation	Support at Grofers	Expectation	Support at Grofers
Ease of Development	**Development Infrastructure**	- Can be manually prepared. New team members find it impossible to get up and running by themselves. Always **need another person on the team to hand hold**.	- README with the description and basic information of the service must be present. Incomplete README is okay but not having a README is not okay. Also refer C24	- Can be **setup in maximum a couple of hours** using repeatable scripts or instructions as documented in the README. **Doesn't need handholding by another person on the team**.	- Steps for installing dependencies, setup and running tests are documented in project's README - Depedency installation and application runtime is dockerized - Config file templates exist in the project repo and working config can be pulled from an already running environment Also refer E24	- Can be **setup in minutes** using containerized infrastructure. Should resemble production. If needed, develop against an integrated environment.	- Single command to setup an integrated debugging environment for locally running application via **telepresence or skaffold** - Configuration should be automatically generated from Consul instead of being manually entered using the same config-templates as for other envs - **Kustomize** is used for overriding any dev/local specific config - Helm is used with Skaffold for setting up open-source services like postgres, redis, mongo, etc. - Dev friendly Dockerfiles. Can we save state of docker work somehow? Can we pipe commands from terminal to docker container so that bash history is not lost? .dockerignore to support thin docker images locally.	- **Fully integrated environment for local development is not needed**. Development can reliably happen against **stubs/mocks/contracts (also see contract testing in the "Testing" section)**.	- Services and dependecies can be setup locally with one step - Critical depedencies should be mocked using pact/mountebank/etc so that service can indepedently come up. Apparantly API gateways can also be used for mocking.
Releasability	**Configuration Management**	- Secrets are stored in an external secret storage system and not committed in SCM - Configuration files are templatized and config values are version controlled per environment - Config is version controlled with the application code - Every config change needs a manually triggered deployment.	- Secrets are stored in Vault - Applications get secrets using HVAC, consul-template - Configuration is stored in Ansible repos or Consul. - Applications get configuration using consul library or consul-template. - KVs in Consul are version controlled using ConsulConfigMap - How do we streamline config deployment? It's broken and differently done everywhere right now. Explore consul-k8s	- Configuration management across multiple envs (prod, stage, CI, dev) is easy	- Use Kustomize with repositories for manging environment specific configuration and manifests (including KVs in Consul using ConsulConfigMap). Uses Ansible to achieve the same thing for workloads on EC2.	- Every system config change triggers a graceful application reload to pick up new configuration change instead of a full deployment. - Every application config change can reload the cofnig context of the application (i.e. doesn't require an application process restart).	- Applications gracefully handle config updates by auto-reloading applications using consul-template. - Application config is managed in consul, differently as compared to how we manage system configuration. Applications can defined TTL for config expiry according to their business requirements so that not every app config change requires the service to be restart.	- Uses dynamically generated secrets with attached lease for periodic secret rotation. - Applications use IAM roles instead of secrets for IAM users	- Secrets with dynamic credentails are requested using consul-template by applications from Vault. Secrets will have leases attached. Leases get auto-expired. Should be done wherever possible. - Applications gracefully handle expirty of secrets with. leases by auto-reloading applications using consul-template. - kube2iam or an alternative for this
	Build and CI	Automated build and testing. Any build can be re-created from source control using automated process on-demand. Test/build environment could be shared, making it hard to run tests in parallel.	- Use declarative Jenkins pipeline (i.e. Jenkinsfile) to setup CI pipeline. - Docker images with standardized tags are created and pushed for git pushes for server apps. Refer this for current standard - https://grofers.atlassian.net/wiki/spaces/RELENG/pages/1501593601/Best+Practices#Docker-tagging - Docker images are built and pushed. Can be implemented using CI Conductor. Language ecosystem specific support can be used (for example, standard process for Python, Java, etc.). Artifact repositories can be used to persist builds. - What about frontend clients like andoird app and web?	- Automated build and test cycle (with more types of tests) every time a change is committed (on every push to a pull request). Dependencies managed through code. Re-use of scripts and tools. Tests are run in isolated environments for every PR, making it possible to run tests for changes in parallel. - Automated tests / checks are required to pass before a pull request can be merged. Also refer to E7 and E8	- The CI pipeline should follow this high level standard - https://github.com/grofers/gr-project-template/blob/master/docs/ci-conductor.md#the-pipeline-structure - CI pipleines should create new environment for every PR. Use mft on Kubernetes. For services on EC2, this is not possible and hence these services should migrate to Kubernetes.	Build metrics gathered, made visible and acted on. Builds are not left broken	CI pipeline should emit following metrics - - CI health: - Stability rate of env setup - Speed of orchestration of CI environment (Monitoring needs to be built, more metrics will be added. See this example)	Teams regularly meet to discuss integration problems and resolve them with automation, faster and better visibility	- CI pipeline/practice health reports are reviewed every sprint within the team during retros. Reports should be readily available to every team.
	Environments and Deployments	Automated deployments to some environments (prod and stage,	Automated deployments to some environments (prod and stage,	Fully automated, self service push-button process for	- Use common Jenkinsfile and Kubernetes manifests with	Orchestrated deployments managed. Release and rollback	- Deployment is triggered automatically on creating a	Deployments, promotion and rollback on the basis of SLOs and	- Rollout, promotion, rollbacks done using something like

Platform Roadmap

But do maturity models work?

Thank You!

Questions?

VAIDIK KAPOOR

twitter.com/vaidikkapoor
linkedin.com/in/vaidik
vaidik.in