Kubernetes運用者が開発に入って見えた、開発と運用をつなぐ基盤の価値

Cloud Operator Days Tokyo 2026

新谷 翔 / NTTドコモビジネス

経歴紹介

つなごう。驚きを。幸せを。
docomo Business

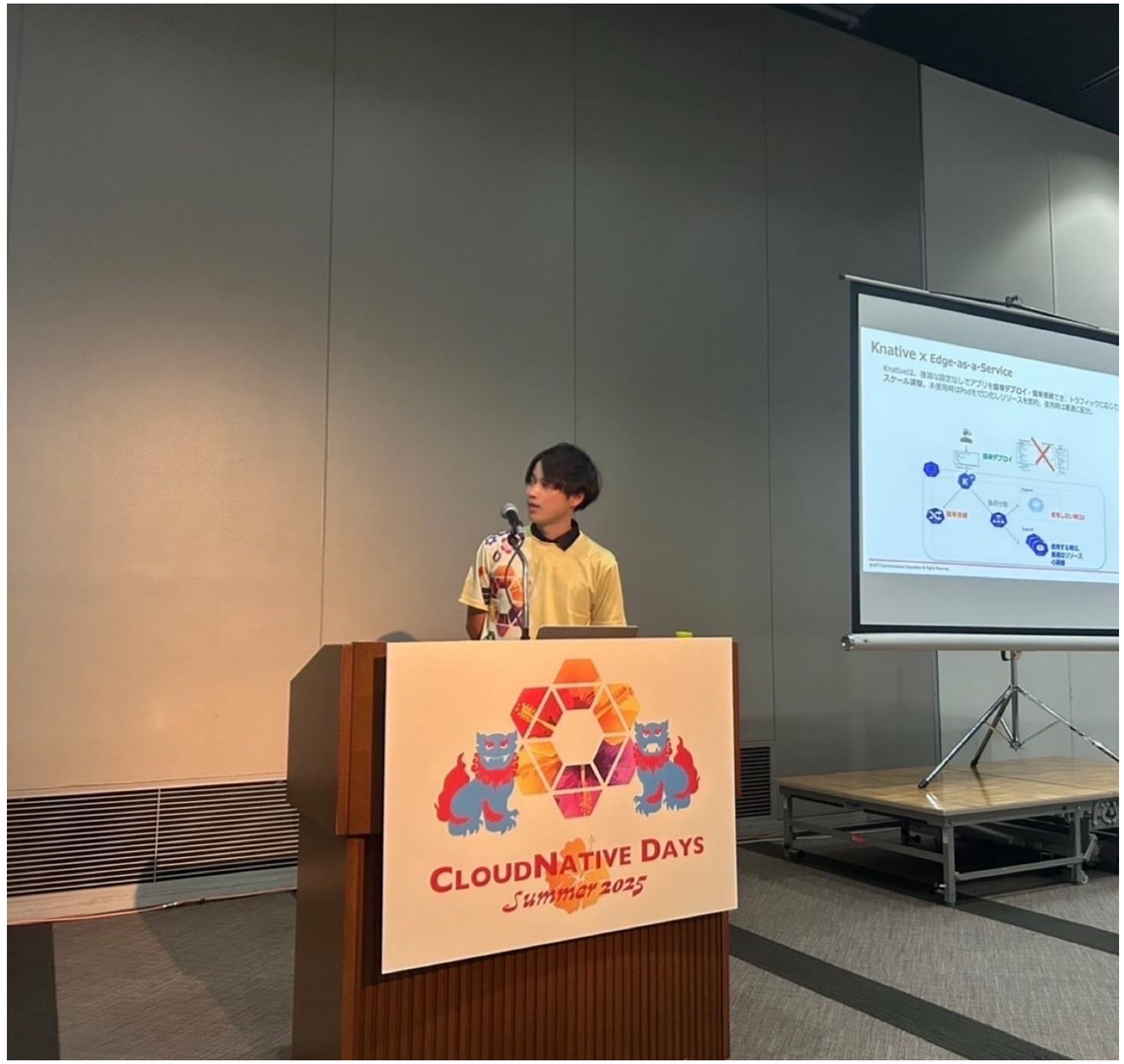
所属
NTTドコモビジネス
プラットフォームサービス本部
クラウド＆ネットワーク部

Before：インフラ・基盤構築

- Kubernetes / Linuxサーバ / ネットワークを中心に基盤構築・技術検証を担当
- 他部署・企業とのPoC、商用化判断に向けた技術評価に貢献
- カンファレンス登壇、特許検討などR&D寄りの業務も経験

Now：商用サービスのAPI開発・運用

- クラウド＆ネットワーク部へ異動後、SDPF上で提供されるセキュリティサービスであるFRA、FSGの開発を担当
- API 開発および運用を担当

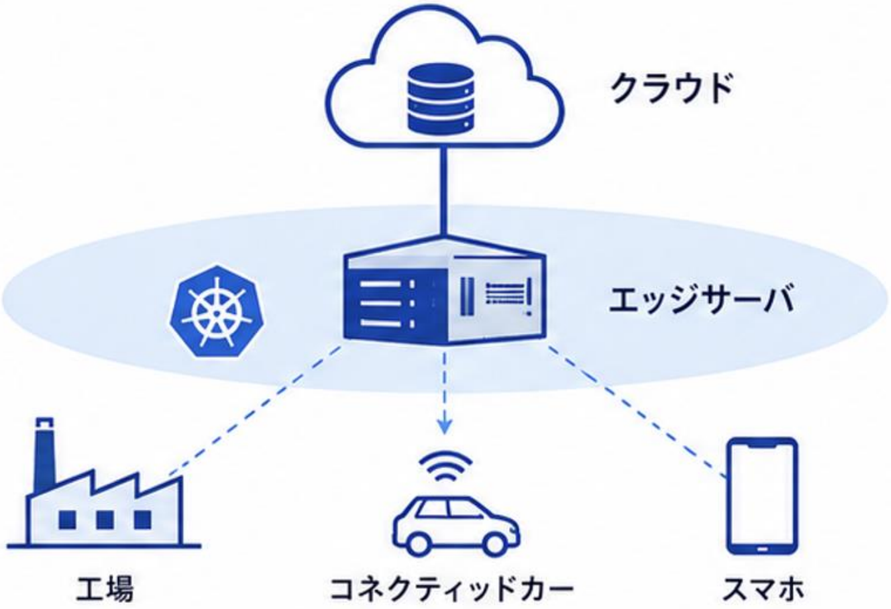


異動する前の業務

つなごう。驚きを。幸せを。
NTT docomo Business

クラウドと端末の間にあるエッジ基盤の構築・検証を担当し、配置、接続、遅延、閉域性、安定稼働といった“基盤を成立させる観点”を主に見ていた。

当時見ていた世界



クラウドと端末の間にある基盤をどう成立させるか

基盤を成立させるための観点

Placement	どこに配置するか
Connectivity	どう接続するか
Resource	どのリソースを割り当てるか
Monitoring	どう監視するか
Stability	どう安定稼働させるか

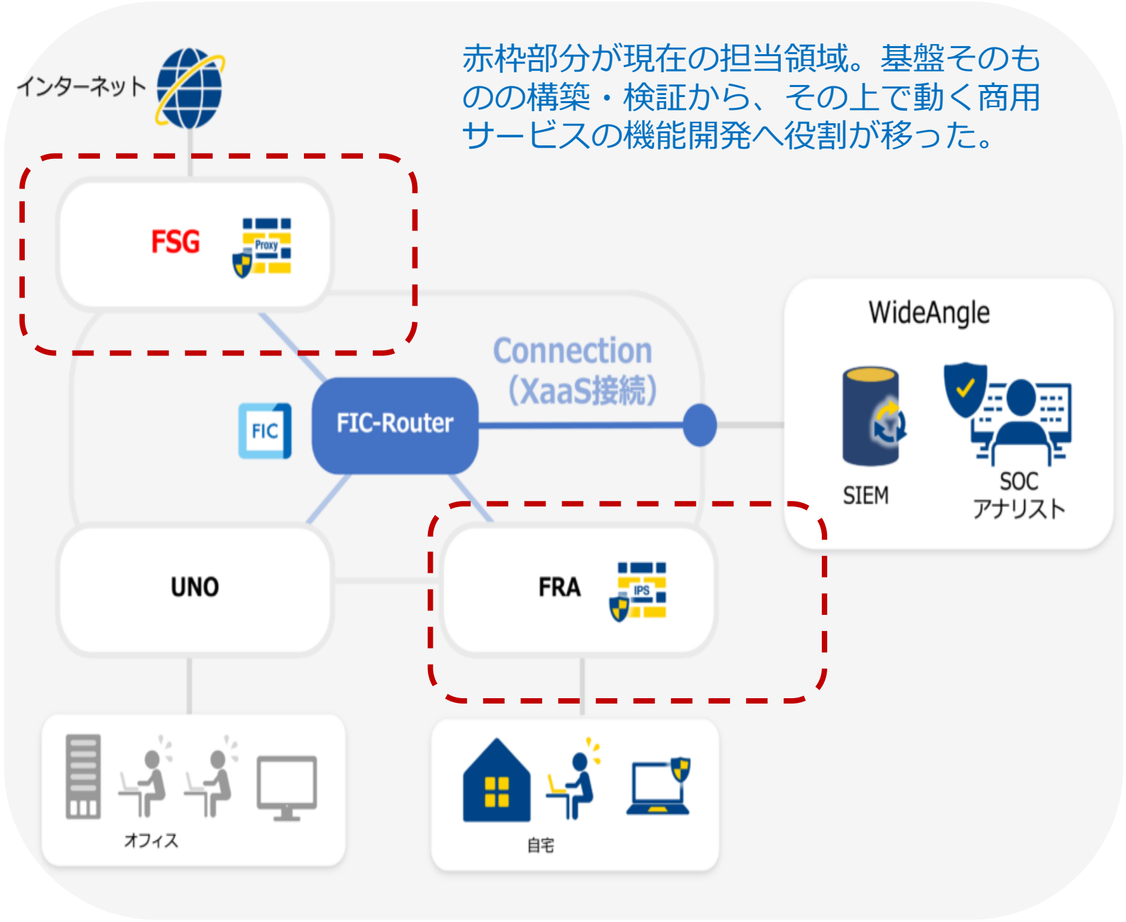
当時の判断軸

- クラスタは安定して動くか
- ネットワーク経路は成立しているか
- リソースは足りているか
- 監視・アラートで状態を追えるか
- 障害時に基盤として復旧できるか

異動後の業務

つなごう。驚きを。幸せを。
docomo Business

現在は、SDPF上で提供されるセキュリティ系サービス（FRA/FSG）の機能開発を担当しています。基盤そのものの構築・検証から、その上で動く商用サービスのAPI実装、試験、運用影響確認までを見る立場へ変わりました。



- 要求・仕様の整理
- OpenAPI / API仕様の確認
- 実装方針の検討
- API実装
- 正常系試験 / 異常系試験
- 権限制御・監査ログ確認
- デプロイ / 動作確認
- 運用影響確認（監視・通知・切り戻し）

引用元:https://sdpf.ntt.com/services/fsg/connected-service/

何が変わった？

つなごう。驚きを。幸せを。
docomo Business

Kubernetes基盤を見ていた立場から、商用サービス開発の視点へと変わった

インフラ側（以前の自分）

主に「基盤」を見ていた

インフラ基盤の構築・運用
Kubernetes / ネットワーク / ストレージ など

可用性・性能・安定性の維持
監視 / アラート / SLO / キャパシティ管理

セキュリティ・ガバナンスの適用
RBAC / ネットワークポリシー / パッチ管理

コスト・リソースの最適化
リソース設計 / コスト管理 / 効率化

標準化・自動化の推進
Terraform / Helm / CI/CD基盤 など

主な関心・判断軸

- 基盤は安定しているか
- リソースは足りているか
- セキュリティ要件を満たしているか
- コストは最適か
- 自動化・標準化できるか

「基盤を安定して動かすこと」がミッション

異動を機に
API開発の担当へ

見るレイヤと
責任がシフト

開発側（現在の自分）

基盤の上で「機能・サービス」を見るように

API・機能の設計 / 実装
仕様整理 / エンドポイント設計 / 実装

データ・権限制御の設計
認証・認可 / テナント設計 / データ整合性

品質保証（テスト・異常系）
正常系 / 異常系 / 結合テスト / 負荷テスト

監査・ログ・トレーサビリティ
監査ログ設計 / 運用ログ / 追跡性の確保

リリース・運用影響の見極め
リリース計画 / 切り戻し / 既存機能への影響

主な関心・判断軸

- 仕様として正しいか
- 権限境界を守れているか
- 異常系は十分か
- 監査・説明責任を満たせるか
- 運用に与える影響はないか
- 安全にリリースできるか

「事業に耐える品質でリリースすること」がミッション

アプリ開発者として触ることで改めて分かったKubernetesの価値

つなごう。驚きを。幸せを。
docomo Business

実行基盤として見ていたKubernetesが、開発に入ると再現性・試しやすさ・流れの一貫性を支える土台として見えた

インフラ側で理解していた価値

実行基盤としての価値を発信

- コンテナを安定して配置できる
- Pod を増減しやすい
- デプロイや監視を標準化できる
- 基盤として安定運用しやすい

見えていたのは、
「クラスタをどう動かすか」だった

理解

↓

体感

Kubernetesの価値が“仕組み”から“開発体験”へ変わった

開発側で体感した価値

Kubernetesベースの開発環境 → **必要な Pod を利用** → **CI/CD と連携**

環境差分を意識しにくい
ローカルだけで完結せず、
実行環境に近い形で確認できる

必要な構成をすぐ試せる
Pod や周辺コンポーネントを
前提に動作を確認しやすい

実装後の流れまで見える
テスト・デプロイまで含めて
変更の通り道を意識できる

強く意識せずに恩恵を受ける
毎回クラスタ操作をしなくても、
再現性と一貫性が支えられている

字面で理解していた価値を、アプリ開発者として初めて“体感”できた。

Kubernetesがありがたかった理由①

つなごう。驚きを。幸せを。
docomo Business

インフラを毎回立ち上げなくても、開発者は“アプリを書くこと”に集中できる。Kubernetesはそのための前提条件を静かにそろえてくれる。

開発者から見た価値

① サーバを毎回用意しなくてよい

ローカルに個別の実行基盤を都度組まなくても、既に用意された環境で開発を始められる。

② 実行環境が整っている

必要な設定・依存関係・接続先があらかじめそろっており、環境差分を意識しにくい。

③ Pod / Service / Ingress を強く意識しなくてよい

開発者は詳細なクラスタ操作より、必要なアプリ動作の確認に集中できる。

④ アプリ開発に集中できる

インフラの複雑さが裏側へ隠蔽されることで、実装・試験・改善のサイクルが速くなる。

Kubernetesベースの開発環境

“複雑さを見せない”ことで開発体験を支える

Kubernetesの価値は“触ること”より、開発者が複雑さを意識せずに再現性と一貫性を得られることにあった。

Kubernetesがありがたかった理由②

つなごう。驚きを。幸せを。
docomo Business

開発からリリースまでを同じ土台に乗せることが可能になる

同じ土台の上で流れる開発ライフサイクル

仕様確認 → テスト → シナリオ → CI/CD → デプロイ が分断されにくい

Kubernetesベースの共通土台

同じ土台だから流れで見える

開発者として感じたありがたさ

開発・試験・デプロイが共通の基盤上でつながる

仕様確認が浮かない
OpenAPIの確認が実装前の作業で終わらない

確認の流れを切り替えにくい
UnitTestやシナリオ実行が同じ文脈で続いて見える

リリースまでの通り道が見える
CI/CDやデプロイを開発の外側に感じにくい

機能確認に集中しやすい
環境ごとの差や手作業に気を取られにくい

開発・試験・デプロイが共通の基盤上でつながることで、機能確認に集中できた。

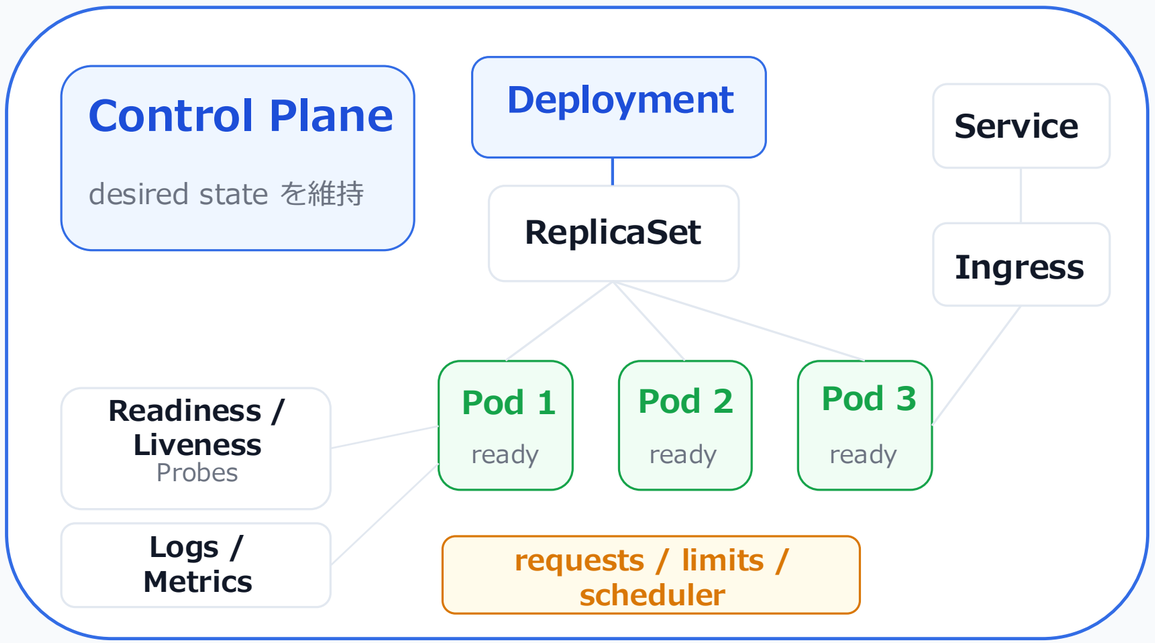
Kubernetesがありがたかった理由③

つなごう。驚きを。幸せを。
docomo Business

Kubernetesは運用に必要な仕組みが最初から標準化されている

Kubernetes primitives

開発者が直接作らなくてよかった「継続稼働の仕組み」



What changed for developers

「構築対象」だったものが、開発では“前提機能”に見えた

Self-healing	Pod failure 時に自動で再作成される 障害復旧の土台を毎回実装しなくてよい
Controlled rollout	Rolling update / rollback を標準で持つ 変更を段階反映し、戻しやすい
Resource governance	requests / limits で資源消費を制御 暴走しにくく、実行時の不安が減る
Observability hooks	probe / logs / metrics を接続しやすい 状態確認と切り分けに入りやすい

Kubernetes は、アプリの外側に自己回復・段階反映・観測可能性を持つ “運用の制御プレーン”

Kubernetesを動かすだけでは見えなかった、商用品質の論点

つなごう。驚きを。幸せを。
docomo Business

インフラからAPI開発へ横断したことで、商用サービスに必要な設計の本質が見えた

Before | 基盤中心で見ていた価値

“動かすための基盤”として主に見ていたもの

Placement	どこに配置するか
Connectivity	どう接続するか
Deployment	どう配るか
Monitoring	どう監視するか
Stability	どう安定稼働させるか

＝ 基盤を成立させるための視点

視点の転換

“どう動かすか”から
“安全に変更を届けられるか”へ

見る対象が
『資源』から『変更の意味』へ
広がった

After | API開発で初めて見えた価値

商用サービスとして“安全に出せる形”へ整えるための設計

API Contract	request / response / status code / schema
Data Boundary	tenant / cell / resource scope
Failure Design	validation / not found / upstream timeout / retryability
Authorization	誰が何を変更できるか
Auditability	後から追えるか・説明できるか
Operational Impact	監視・通知・切り戻し・既存影響

＝ 変更を安全に商用へ出すための視点

横断して得られたもの

- Kubernetesの真価は、アプリを安全に商用へ載せるための“運用可能性”を可視化することだと分かった
- API仕様・権限制御・障害設計・監査性まで含めて初めて“出せる品質”になると理解した
- インフラ観点とアプリ観点をつなぎ、変更影響を前提に設計できるようになった
- “作れる”だけでなく、“説明できる・運べる・戻せる”まで設計する視点を得た

“リリースできる形”とは何か

つなごう。驚きを。幸せを。
NTT docomo Business

最初は、APIを実装して正常系が通れば前に進めると思っていた。しかし、商用サービスでは、“動く”ことと“出せる”ことは別であり、その間には、仕様、異常系、権限、監査、運用影響、リリース導線といった運用設計があった。

でも、それだけでは商用に出せない

“動く”と“出せる”の間に、運用設計がある

1. API仕様
request / response / status
code / schema を説明できるか

2. 異常系設計
validation / not found /
upstream error の挙動が決まっているか

3. 権限制御
tenant / cell / resource 単位で
操作範囲が守られているか

4. 監査性
誰が何をしたか、
あとから追えるか

5. 運用影響
監視・通知・切り戻し・
既存機能影響を確認できるか

6. リリース導線
stgで確認し、
prdへ安全に昇格できるか

そこで見えたKubernetesの価値

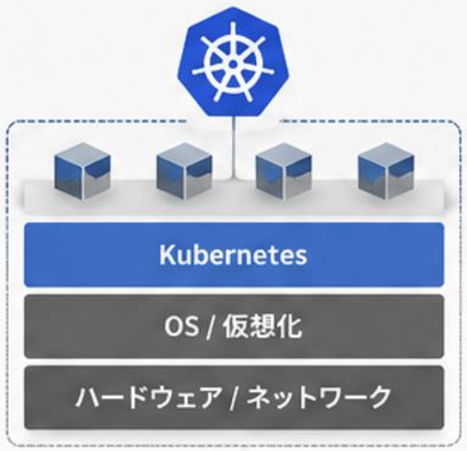
つなごう。驚きを。幸せを。
docomo Business

インフラからAPI開発へ横断したことで、商用サービスに必要な設計の本質が見えた

以前の見え方

基盤を構築し、安定稼働させること

- クラスタ構築・運用
- リソースの配置・拡張
- 可用性・性能の確保



目的：基盤を“動かす・支える”

今の見え方

変更を安全に届け、運用できる状態にすること

目的：サービスを“安全に変更し続ける”

事業・サービスへの価値

安全性の向上
段階的なリリースや切り戻しにより、
リスクを抑えて提供できる

開発スピードの最大化
標準化されたプラットフォームにより、
チームは価値開発に集中できる

信頼性・可用性の確保
自動復旧やヘルスチェックにより、
安定したサービス提供を支える

運用負荷の最適化
可観測性と自動化により、
運用判断を迅速かつ正確に行える

ガバナンス・コンプライアンス
権限制御や監査ログにより、
説明責任と内部統制を満たす

AI時代ほど、“出せる品質”を判断する力が問われる

つなごう。驚きを。幸せを。 docomo Business

AIが速くするのは実装。商用に出せる品質へつなぐのは、Kubernetes基盤と人の判断だ。

① AIが速くするもの

コード生成
API / UnitTest / 雛形の初版を
素早く作れる

仕様理解の初速
OpenAPI・設計書の読み解きや
論点整理を早められる

調査・比較
関連実装・エラー・ドキュメントを
素早く洗い出せる

文書化
変更説明やレビュー用の
たたき台を短時間で作れる

“書く速さ”は上がる

② Kubernetes基盤が効く理由

再現性のある開発環境
ローカル差分を減らし、同じ前提で試せる

必要なPodをすぐ利用
実行環境や周辺コンポーネントをすぐ呼び出せる

CI/CDと同じ土台
開発・試験・デプロイの流れがつながる

観測しやすい
ログ・監視・リソース状況を追いやすい

安全に回しやすい
ロールアウト / 切り戻し / 再作成が前提にある

実装 → 検証 (Pod) → CI/CD (Pipeline) → Deploy (Deployment) → Observe (Monitoring)

Kubernetesは“インフラの構築対象”ではなく、
“実装を速く・安全に回すための共通基盤”として効く

③ それでも最後に人が判断するもの

API Contract
その request / response / status で
本当に妥当か

Authorization
tenant / cell / resource の
権限境界を壊していないか

Failure Design
異常系・upstream error・timeout 時の
挙動が決まっているか

Operational Impact
監視・通知・切り戻しを含めて
運用に耐えられるか

Release Decision
既存機能影響を踏まえ、
商用に出してよい状態か

“出せる品質”は人が決める

AI時代に価値になるのは、単に速く実装する力ではない。
Kubernetes基盤を活用して早く試し、品質・運用影響まで含めて“リリースできる形”へつなぐ力である。

まとめ

つなごう。驚きを。幸せを。
docomo Business

AIがコード生成を支援する時代だからこそ、エンジニアの価値は
「何を作るか」「どこにリスクがあるか」を判断し、事業に耐える品質へつなぐ力 に移っている

これまで（Kubernetes運用）

システムを安定して動かすことがミッション

- **基盤の設計・構築・運用**
  Kubernetes / IaC / CI/CD
- **可用性・信頼性の向上**
  監視 / アラート / SLO / DR
- **セキュリティ・ガバナンス**
  RBAC / ネットワーク / ポリシー
- **コスト・リソース最適化**
  キャパシティ / コスト管理
- **標準化・自動化**
  テンプレート / 運用手順 / 自動化

開発に入って見えたこと

「作ること」だけでなく「リリースできる形にする力」が必要

仕様整理・合意形成 要件を整理し、API契約に落とし込む	**権限制御の設計** tenant / cell / resourceの境界を正しく設計	**監査ログ・トレーサビリティ** 誰が・いつ・何をしたか追える設計と実装
異常系試験・品質保証 バリデーション / エラー系結合・負荷・回帰テスト	**運用影響の見極め** 既存機能・運用・コストへの影響を事前に評価	**リリース・運用準備** ロールアウト戦略 / 切り戻し監視・アラート整備

今の自分の結論

開発と運用をつなぐ視点が、
事業に耐える品質を生む

- 基盤とアプリの両方を理解し、
  変更のリスクを立体的に見極められる
- 「安全に変更を届ける仕組み」を意識して、
  開発スピードと安定性を両立できる
- チーム・事業の信頼をつなぎ、
  継続的に価値を届けられるエンジニアになる

これからの自分の軸

仕様・リスクを見極める → 正しく作る → 安全に届ける（運用までつなぐ） → 事業価値につなげる

Kubernetesで培った基盤の知見と運用の視点は、
API開発の品質とスピードを支える土台になる。
これからも、開発と運用をつなぐ架け橋であり続けたい。

つなごう。驚きを。幸せを。

NTT docomo Business