GraphQL Blue-Green Deployments

Speaker: Zack Warnimont

What We'll Cover

- The Blue-Green deployment pattern
- Risky GraphQL schema changes
- Canary deploys without two separate graphs
- Schema versioning with OCI distribution
- How tag-based routing makes rollouts and rollbacks trivial (and safer!)

What's the Problem?

- Current system was nontrivial to roll back without deployment
- Existing functionality was insufficient (e.g. percentage rollouts on GraphQL directives)
- Directive changes require a deploy. Slow rollback, manual increments

What is a Blue-Green Deployment?

- Two environments serving production: Blue (live) and Green (new)
- Incremental percentage* of traffic routed to green
- Flip back to Blue if metrics tank
- Both versions co-exist during deployment/rollout

* - best practice, technically this is "blue-green + canary"

Why is Blue-Green deployment important?

- Up front investment, long term savings
- Deployment safety: reduce outage blast radius
- Testing becomes more important as AI is leveraged to deliver more code quickly

Why Schema Changes Are Risky

- Not applicable: Breaking changes are usually caught by tooling before deploy
- Valid, additive changes can pass checks and cause issues at runtime
- Staging doesn't always accurately represent production volume / latency
- The schema change is "correct", but resolvers are slow / problematic

Example: Personalizing an Existing Field

- You have a User.recommendedProducts field that clients already actively query
- You want to personalize it: rank results using the user's purchase history
- Purchase history lives in the Orders subgraph, use `@external`/`@requires` to contribute computed field
- The schema composes and validates successfully

Adding @requires to an Existing Field

Before (Blue)

```
# Products subgraph
type User @key(fields: "id") {
  id: ID!
  recommendedProducts: [Product]
}
```

After (Green)

```
# Products subgraph
type User @key(fields: "id") {
  id: ID!
  purchaseHistory: [Order] @external
  recommendedProducts: [Product]
    @requires(fields: "purchaseHistory")
}
```

The Problem Shows Up at Scale

- @requires: indicate that provided @external fields are needed to resolve the field
- latency to @external subgraph is low in staging
- unexpected load in production causes latency spike
- recommendedProducts was already being queried. Every one of those queries slows down

Traditional Workaround: Two Separate Graphs

- Maintain canary / prod graphs
- Deploy canary graph, then promote prod graph

Problems with this approach:

- Maintainance burden
- Rollback is a deployment

What We Need

- One graph
- Deployment separate from traffic shifting
- Gradual traffic shifting
- Instant rollback
- Immutable version history: every deployed schema is preserved

Schema Registry via OCI Distribution

- OCI (Open Container Initiative): standard artifact storage and distribution
- The same format used to distribute container images
- A schema version is published as an OCI artifact: immutable, content-addressed
- Every published schema gets a unique content hash
- Human-readable tags are mutable pointers to those hashes

What's in the OCI image?

- GraphOS currently puts supergraph schema in a single layer
- Could potentially store other data, e.g. persisted queries

Tags Point to Schema Versions

- Think of it like container image tags: myimage:latest, sha256:abc123...
- Schema references work the same way:

```
myschema:production  →  sha256:a1b2c3d4…  (current)
myschema:canary      →  sha256:e5f6g7h8…  (new version)
```

- The router is configured to follow a tag
- Tag movement = traffic routing (or "deployment")

How Schema Updates Work

- Multiple options
- Hot reload: Router listens for tag changes, reloads schema when sha changes
- deployment-based: CD listens for tag change, redeploy router with digest-addressed sha

The Deployment Workflow

1. Publish the new schema: it gets a content hash, nothing changes in prod
2. Shift percentage of traffic to new hash
3. Validate: queries work, clients updated, metrics look good
4. Shift all traffic to new hash
5. Automate above: roll back when metrics don't look good

Rollback Without Waiting

- Something went wrong after cutover? Rollback in seconds:

```
# move tags back to previous stable
myschema:production  →  sha256:i9j0k1l2…  (previous stable)
myschema:canary      →  sha256:i9j0k1l2…  (previous stable)
```

- No waiting for CI / deployment time
- The old schema artifact was never deleted

What to Watch During a Canary Deploy

- Resolver latency: are any fields slower than baseline?
- Error rate: are any queries returning errors that weren't before? (timeouts?)
- Generally: if anything looks wrong, move the tag back

Router Fleet Configuration (hot reloading)

- 10% of fleet is assigned to `canary` tag (green)
- 90% of fleet is assigned to `production` tag (blue)
- Tag canary metrics so you can query on new deployment (e.g. `service.error_rate{tag=canary}`)
- Add log dimension of tag/digest in subgraphs so you don't get lost

```
$ GRAPH_TAG=canary run_my_router
$ GRAPH_TAG=production run_my_router
```

Router Fleet Configuration (deployment-based)

Tag by environment, roll out tag updates. e.g.:

1. OCI production tag is updated upstream (e.g. from federation provider)
2. Your CD picks up the tag change (from event or polling)
3. CD writes the tag's new sha to your router deployment manifest
4. CD rolls out the sha procedurally (e.g. 1 pod, 2 pods, 4...)
5. CD checks metrics and rolls back when they exceed thresholds

```
$ GRAPH_DIGEST=a1b2c3d4 run_my_router
```

Why the OCI Pattern Works Well

- Instant rollback without deployment
- Immutable version history: every schema published is preserved
- One graph, one router: no infrastructure duplication ➡ no extra maintenance
- Familiar mental model: container image tags translated to schema tags

When to Use Blue-Green for GraphQL

- As often as possible (reduce outage blast radius)
- When automated, generally only serves as a benefit

Difficulties:

- If not automated, manual verification is a slowdown
- Ensuring backwards compatibility is a slowdown, but non-compatible changes are inherently risky

Key Takeaways

- Schema validation catches breaking changes, but valid changes can still hurt in production
- OCI distribution is a proven pattern: immutable artifacts, mutable tags
- Moving a tag = deploy; moving it back = rollback
- A canary is your production smoke test

Questions?

Thank you!