our team builds the TAK software. The software has 5 components. nats, ignite, config, api and messaging nats is a go application and deployed as a stateful set ignite is an apache product and is deployed as a stateful set and deployed as a stateful set config, api and messaging are spring boot applications and deployed as deployments Ignite is a caching service when a user uses the application, they service calls from the api and messaging service are registered with a specific ignite pod if that ignite pod goes down, all the users connected to that ignite pod are pointing to a non existent pod, and the application crashes how do i rearchitect my product to work around it

Thanks for the context — this is a classic case of tight coupling to a specific Ignite node, which goes against Ignite's design goals as a distributed system.

To make your TAK system more resilient and avoid user sessions crashing when an Ignite pod goes down, you need to decouple the application's dependence on a specific Ignite pod. Here's a structured re-architecture recommendation:

✅ 1. Use Ignite in Client-Server Mode (Not Server-Only Cluster)

Ensure your Spring Boot applications (`api`, `messaging`, `config`) connect to Ignite in client mode rather than server mode.

- Current issue: It sounds like your services are "bound" to a specific server node.
- Fix: Use `IgniteConfiguration.setClientMode(true)` in Spring Boot apps so they act as Ignite thin clients that can connect to any healthy server node.

> 🔄 This ensures that if one Ignite pod goes down, the client will automatically reconnect to another node.

✅ 2. Enable Partitioned Cache with Backup Replicas

If a partitioned cache is used (default in Ignite), make sure backups are enabled, so that data is not lost if one pod goes down.

Example in your configuration:

```java
CacheConfiguration<String, SomeObject> cacheCfg = new CacheConfiguration<>("myCache");
cacheCfg.setBackups(1); // Ensures redundancy
```

> 🔁 With this, when an Ignite node (pod) fails, data is still available from another node, and rebalancing happens automatically.

✅ 3. Use Service Discovery Correctly

If you are not already, enable Kubernetes IP Finder or another proper discovery mechanism so Ignite pods can discover each other dynamically even as nodes go up and down.

For Kubernetes:

```java
TcpDiscoveryKubernetesIpFinder ipFinder = new TcpDiscoveryKubernetesIpFinder();
ipFinder.setNamespace("your-namespace");
ipFinder.setServiceName("ignite-service");
```

Attach to `TcpDiscoverySpi` in your Ignite config.

✅ 4. Graceful Client Reconnection Handling

Ensure that the Spring Boot services:

- Retry failed operations
- Handle `ClientDisconnectedException` gracefully
- Possibly implement a retry with backoff logic

✅ 5. StatefulSet vs Deployment for Ignite

Since Ignite nodes have stable identity (Node ID, IP), it's common to use StatefulSet, but it's not strictly required if you handle discovery via service.

> ✅ Stick with StatefulSet, but make sure you're not binding clients (Spring Boot apps) to a specific pod like `ignite-0`. Instead, use a headless service or dynamic IP finder.

✅ 6. Monitor and Auto-Heal the Cluster

Use Prometheus + Grafana or Ignite metrics export to monitor cluster health. Set up Kubernetes probes to ensure unhealthy Ignite pods are restarted automatically.

✅ Optional: Use Ignite Thin Clients for Loose Coupling

The Spring Boot apps can connect via Ignite thin client instead of embedded client node:

```java
IgniteClient client = Ignition.startClient(
    new ClientConfiguration().setAddresses("ignite-0:10800", "ignite-1:10800"));
```

> 🧠 Advantage: Thin clients are lightweight and don't join the cluster, avoiding overhead and tight coupling.

🔧 Summary of Actions

Change	Why It Helps
Use `setClientMode(true)` in services	Avoid hard binding to specific pod
Enable cache backups	Prevent data loss on pod failure
Use Kubernetes IP Finder	Auto-discover nodes
Handle disconnects in clients	Prevent crashes
Use Thin Clients	Optional, better separation
Monitor and Heal	Maintain cluster integrity

Let me know if you'd like a working YAML setup or Spring Boot Ignite config examples.