

The Hitachi logo is displayed in a bold, black, sans-serif font in the top right corner of the slide. The background of the entire slide is a photograph of a landscape at sunset or sunrise, with a dense forest in the foreground and a city skyline visible in the distance under a hazy, orange-tinted sky.

HITACHI

Beyond stateless: Distributed Transaction with Autoscaling and consistency on Kubernetes

About Me

Jumpei Nishitani

Hitachi, Ltd.

Senior Software Architect

Experience:

2007-2012: Development and maintenance of Web server

2012-2018: Development and maintenance of Application Server

2018-2019: Planning of Cloud Migration

2020-2023: Research and development of Microservices Transaction

2024- : Architect Cloud-Native Enterprise System

Contents

This session presents a practical method for implementing distributed transactions with strong consistency and seamless autoscaling on Kubernetes. By integrating a transaction monitor with the Horizontal Pod Autoscaler, we show how stateless business applications can retain ACID properties while scaling efficiently. Based on real-world examples, the session offers actionable insights into system design and deployment.

1. Online Transaction needs and pain point structure of its state machine.
2. How to overcome state machine scaling problem.
3. Transaction processing mechanism of Paxos commit Transaction orchestrator with autoscaling.

The Hitachi logo is displayed in white, bold, uppercase letters in the top right corner of the slide. The background is a solid red color with a large, dark red curved shape on the left side.

HITACHI

Online Transaction needs and pain point structure of its state machine

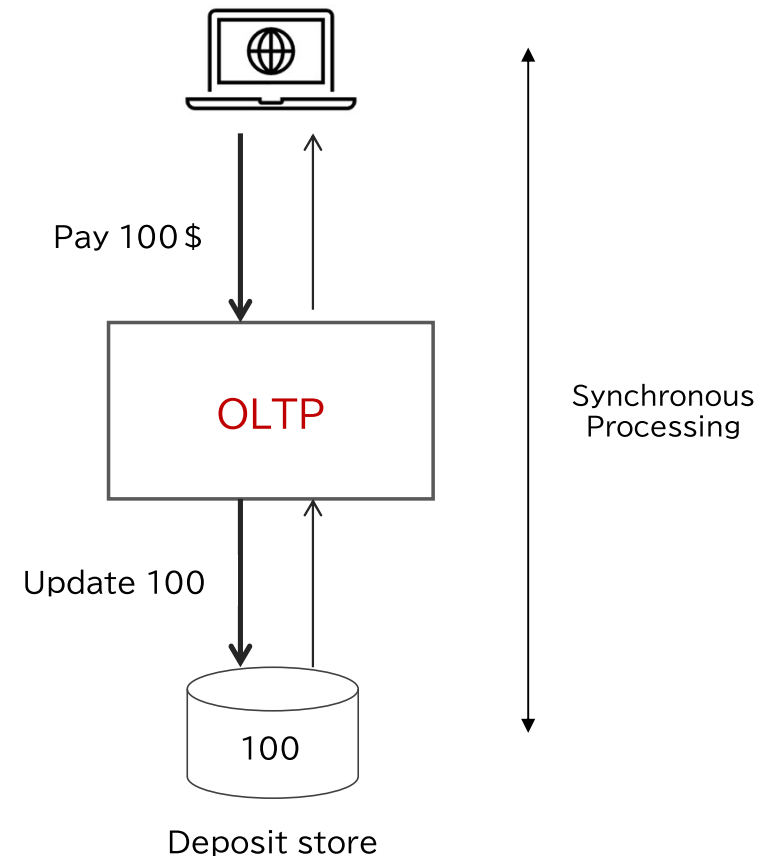
What is the OLTP

Online Transaction Processing

- An IT system that interactive Update Data for transaction like processing a payment, contract, settlement to database.
- In this case, online means the same as interactive or synchronous.

Challenges inherent in OLTP

- Failure in update control can lead to the loss or tampering of business data involving money. This carries significant social implications.
- Despite their significant social impact, updates are vulnerable to system failures and communication delays.



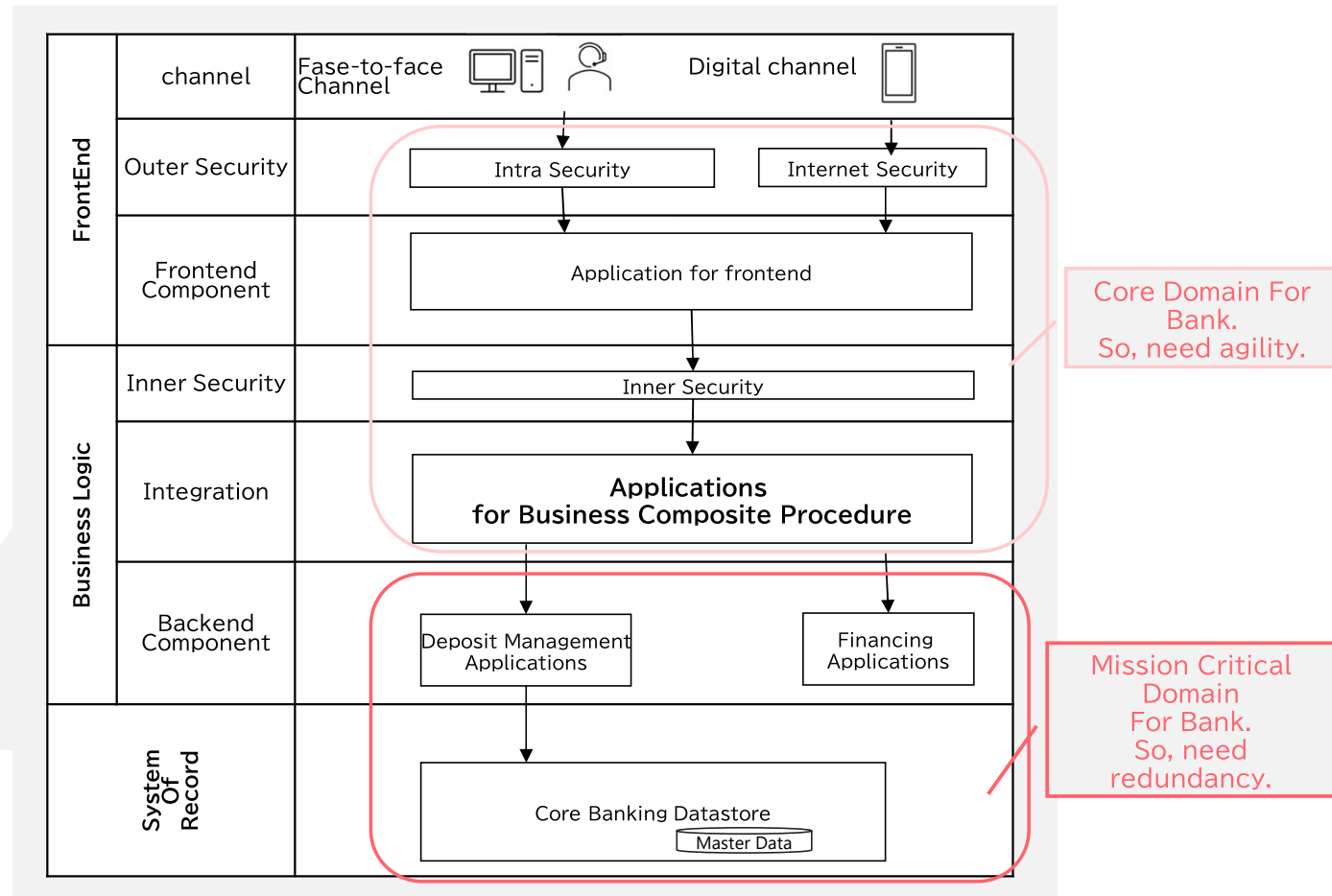
The Ideal of Modern Enterprise System Architecture (Using Banking Systems as an Example)

HITACHI

Enterprise systems are typically divided into the following three layers, and business applications are deployed according to the requirements of each layer.

- **Frontend:** User Experience
- **Business Logic:** Collection of Business Procedure
- **System Of Record:** Record Master Data like account of money

The layers that enhance agility and drive innovation are the “frontend” and “business logic”.



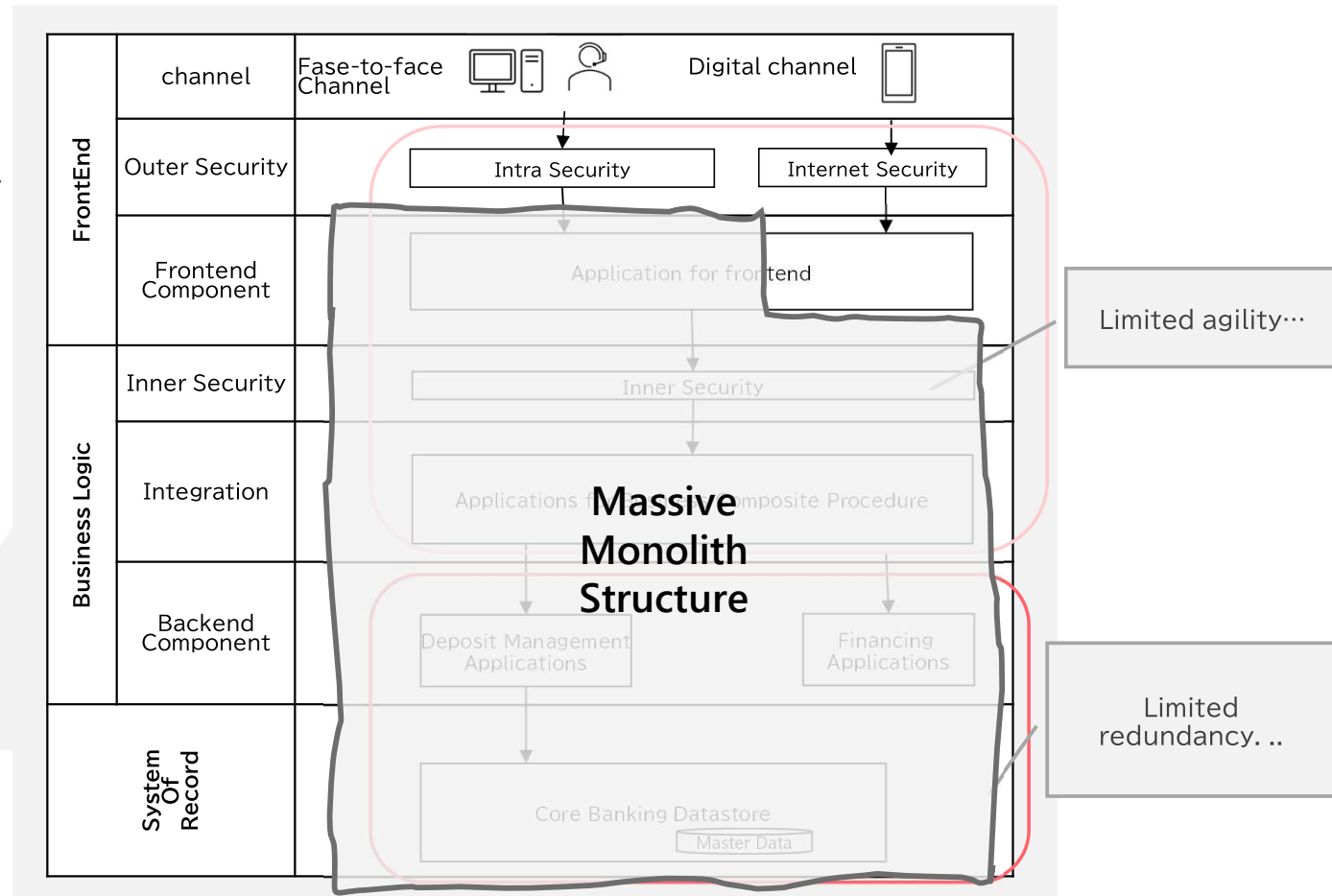
The Ideal of Modern Enterprise System Architecture (Using Banking Systems as an Example)

HITACHI

However, many IT services continue to operate as massive monolithic structures spanning from frontend to backend. Remaining monolithic results in high upgrade costs, relatively low agility and scalability, and hinders innovation. Why do we continue to operate with a monolithic structure? It's because the switching costs to break up the monolith are too high.

Examples of technologies with high switching costs::

Stateful components that cause system outages when they fail like database and transaction monitor



The Hitachi logo is displayed in white, bold, uppercase letters in the top right corner of the slide. The background is a solid red color with a large, dark red curved shape on the left side.

HITACHI

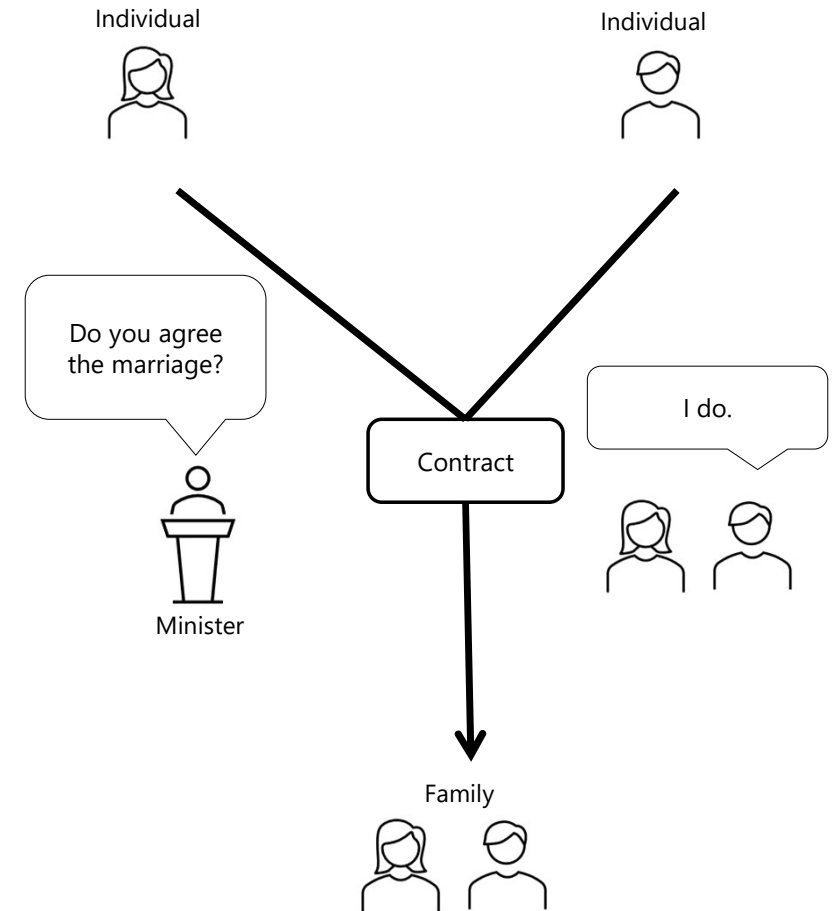
How to overcome state machine scaling problem in Online Transaction

“What is the Transaction?” according to Jim Gray※

The transaction concept comes from contract law.

Parties negotiate and then bind a deal through a formal act, such as signing a document or using an intermediary like an escrow officer. A Christian wedding is a typical example: after prior negotiation, a minister confirms consent and declares the contract complete. Although individuals may violate contracts, legally they can be annulled only if they were illegal, and problems are resolved through compensating transactions.

Translating the transaction concept to the realm of computer science, we observe that most of the transactions we see around us (banking, car rental, or buying groceries) may be reflected in a computer as transformations of a **system state**.



What is State machines ?

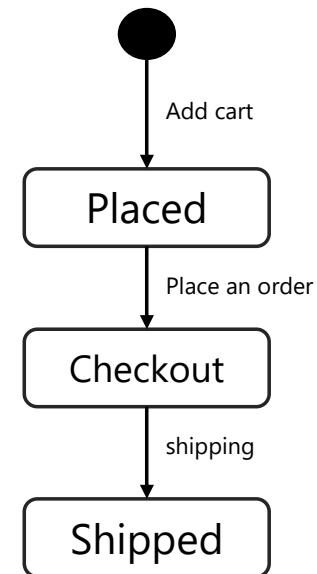
A state machine describes the current situation and what happens next when an event occurs.

For example, an order moves through states such as placed, shipped, and completed, based on clear rules.

Business operations are a collection of rules, so making states clear also makes the rules clear.

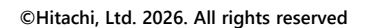
This makes it easier to build IT systems that accurately reflect how the business works.

State Machines is very useful, but it has a weak point.



HITACHI

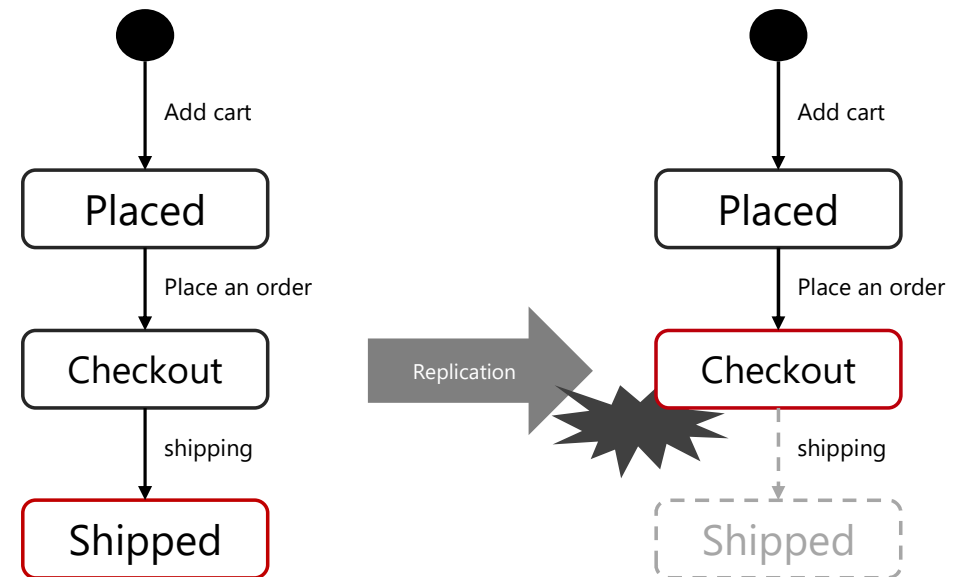
- So, you might think, “Why not just create replicas to handle failures?”



Weakness of State Machines in Distributed Systems

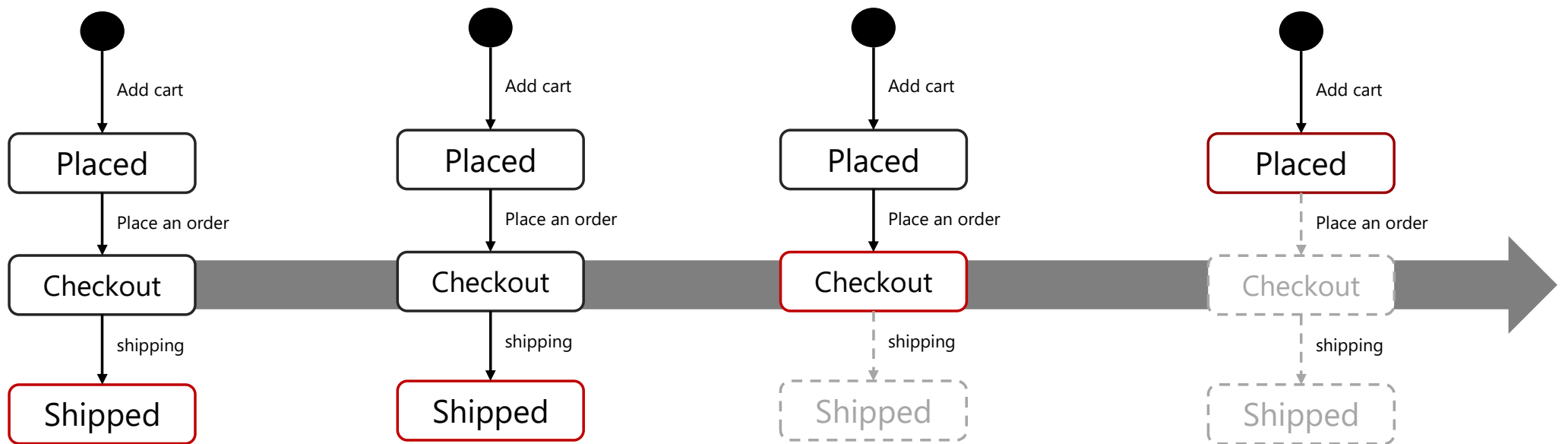
HITACHI

- State machines are also weak in distributed environments
- To handle failures, state machines are often replicated across multiple nodes
- Replication requires messages to keep states in sync
- If message delivery fails, states can diverge between nodes



Weakness of State Machines in Distributed Systems

Even though it's bad at distribution, scaling it out carelessly just makes things worse...

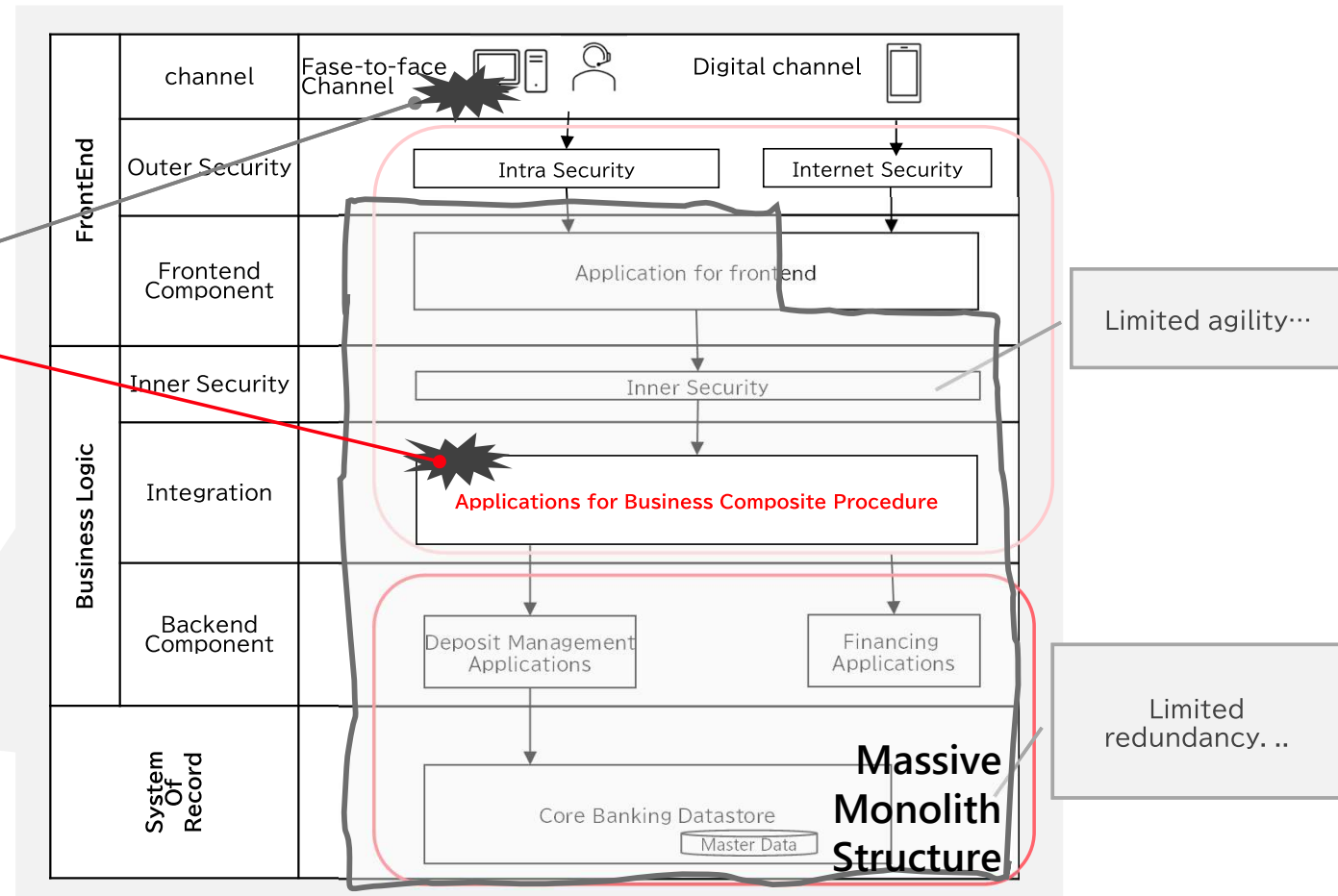


I have no idea what's happened...

The Ideal of Modern Enterprise System Architecture (Using Banking Systems as an Example)

HITACHI

In online transactions, failures directly affect users because the system is being used in real time. As a result, users can suffer serious problems when something goes wrong. For example, if a failure occurs after a customer inserts their card into an ATM, the card may not be returned. Incidents like this can become a serious responsibility issue for the company.



The Ideal of Modern Enterprise System Architecture (Using Banking Systems as an Example)

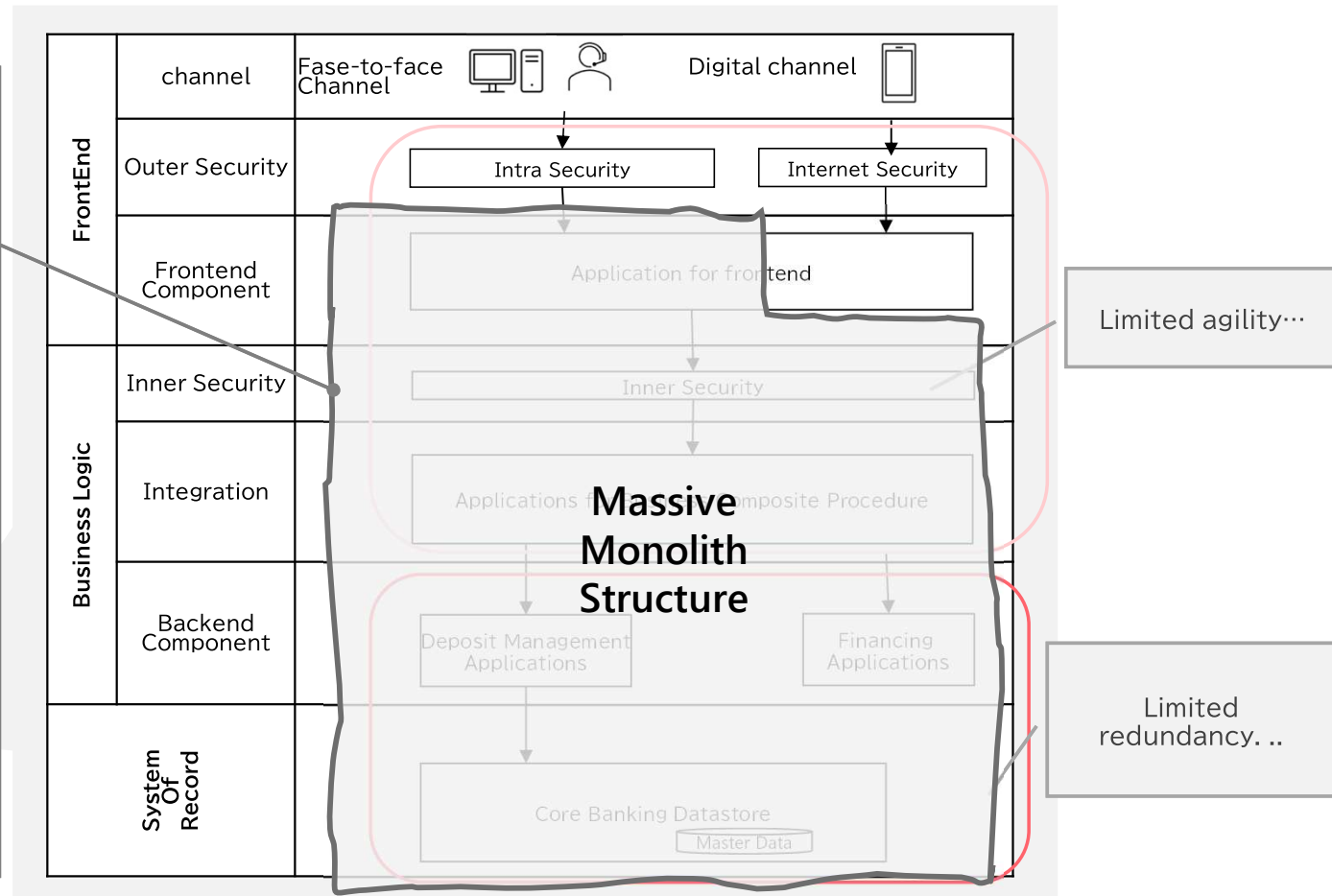
HITACHI

We want to make state machines redundant and scalable to enable scalable procedural applications. However, distributing stateful components can cause a combinatorial explosion due to the synergistic effect of failures and state variations.

Number of State Combination

Number of failure point

Growing beyond hundreds of millions, with no upper limit

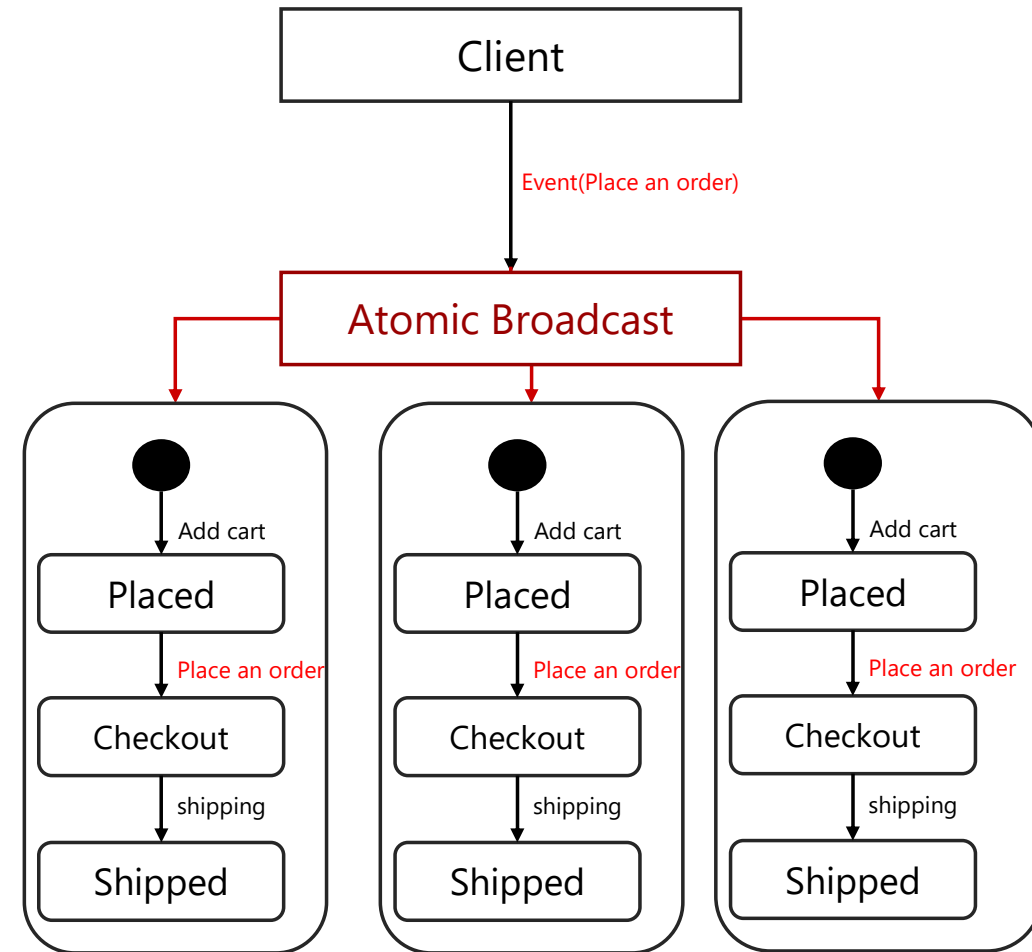


How do we solve the problems of state machines?

HITACHI

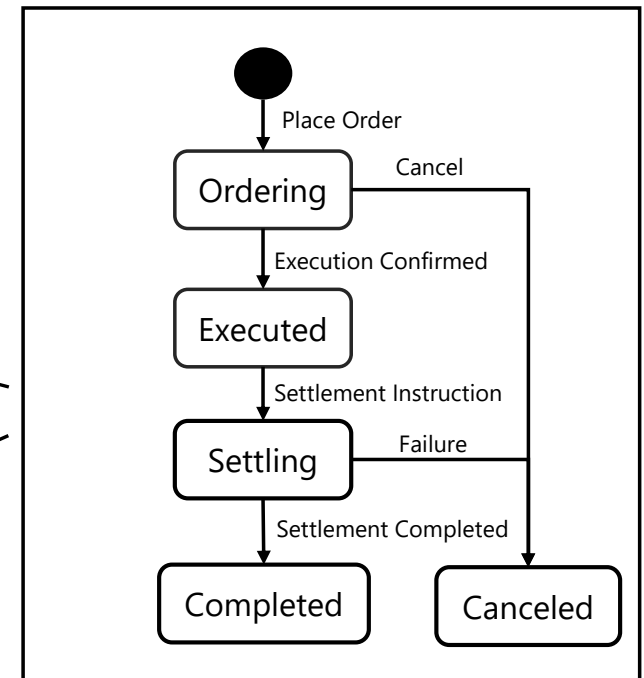
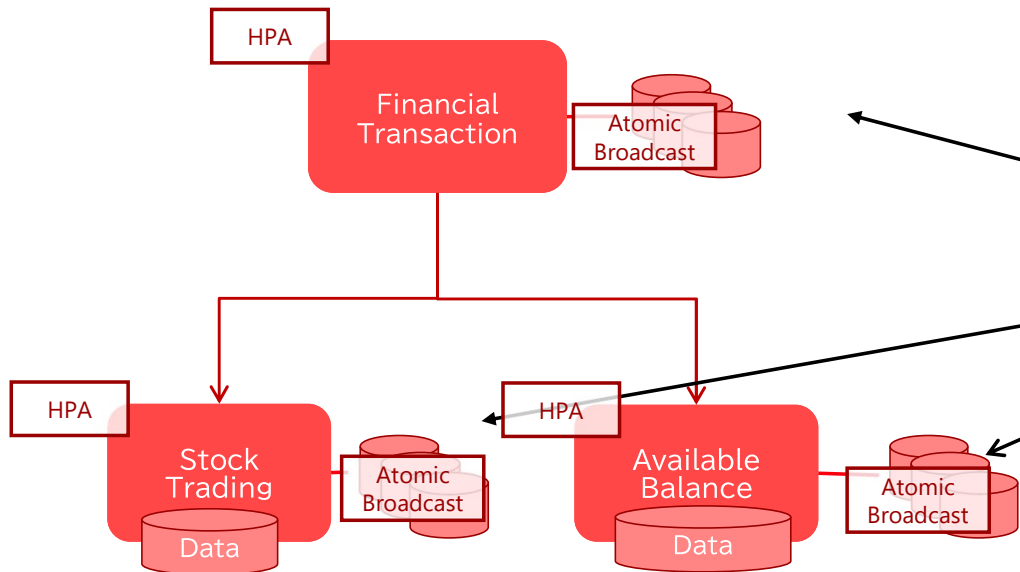
What needs to be done

- Force all nodes to apply state transitions in the same order
- Retry event broadcasts during failures or partitions to preserve ordering
- This is exactly what algorithms like Multi-Paxos, Raft, and ZAB already do
- In practice, using something like Raft-backed etcd for state storage can address most of these issues



How Can We Solve the state Machine Problem?

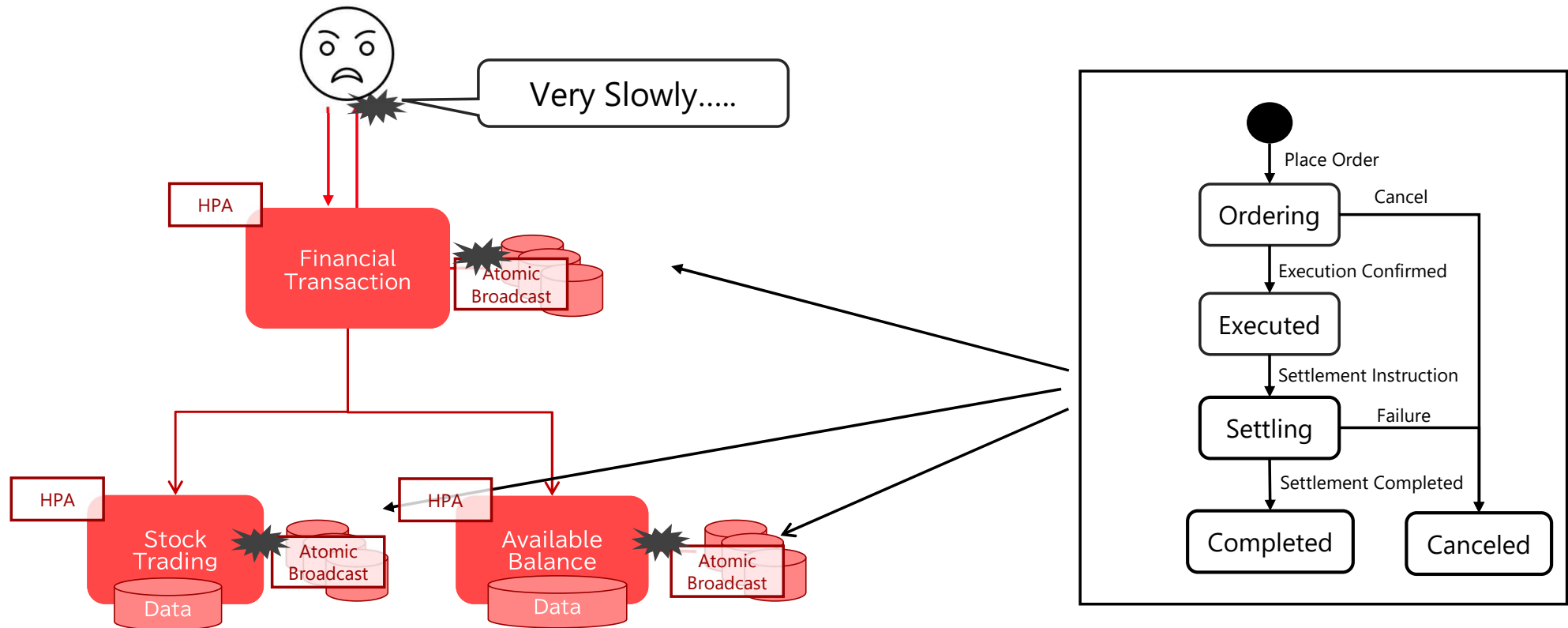
Using a financial transaction as an example, buying a stock completes settlement by exchanging the user's available balance for the stock; if this fails, the transaction is canceled. If all related microservices (Financial Transaction, Stock Trading, Available Balance) share the same state transitions, storing that state in an atomic-broadcast system such as etcd allows the services to be stateless, redundant, and safely scalable.



How Can We Solve the state Machine Problem?

However, this is not so simple. Online transactions are executed while interacting with end users of IT systems, which imposes strict response-time requirements—typically within one second.

In contrast, atomic broadcast is inherently expensive. As the number of microservices increases, the cost of persisting state also increases, which can consume the limited response-time budget and make it difficult to meet those constraints.



When Microservices Stop Scaling: The Hidden Cost of Consensus

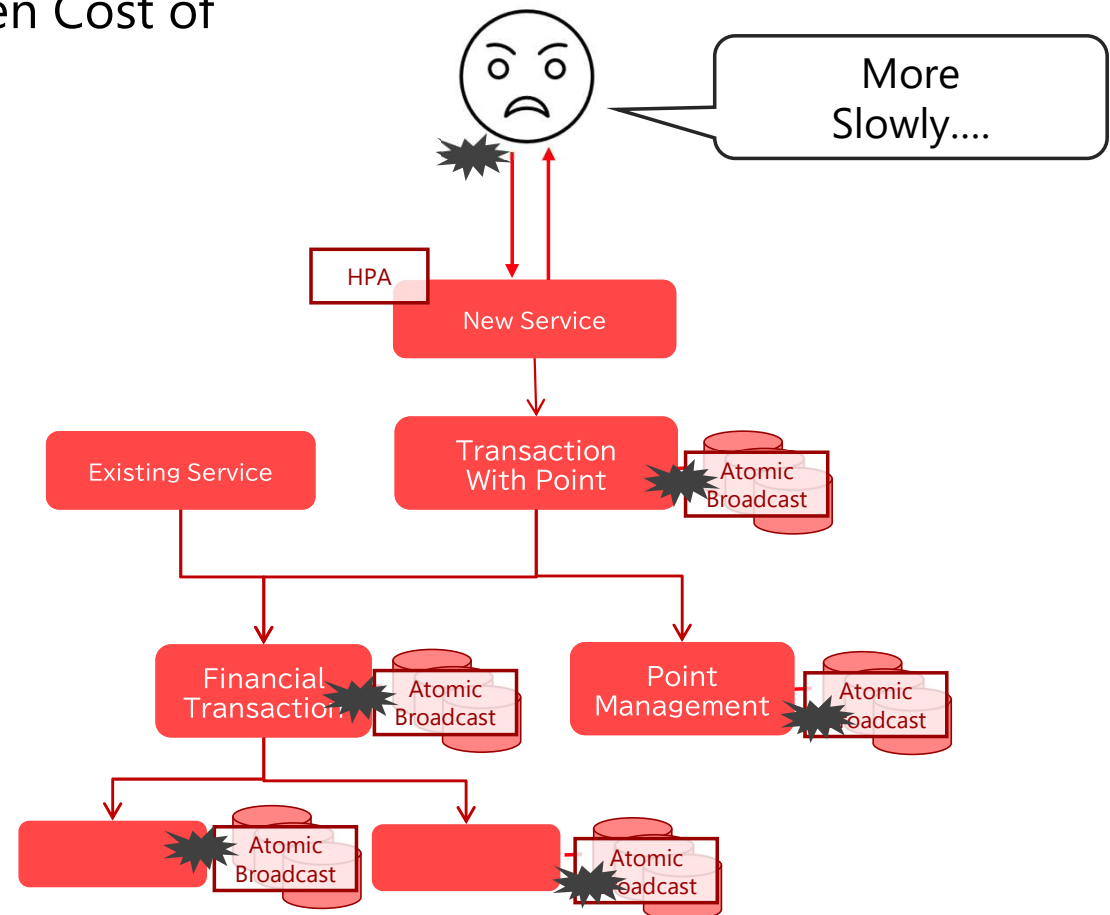
This issue directly affects the scalability of microservices.

As innovation progresses and more microservices are added, the cost of atomic broadcast accumulates. This increasing cost extends response times.

As a result, response-time constraints effectively become the upper limit of microservice scalability. In that sense, it is difficult to say that this architecture offers high scalability as a microservice system.

So what should be done?

Our approach was to step back and revisit the requirements, then redesign the transaction processing model from the ground up.

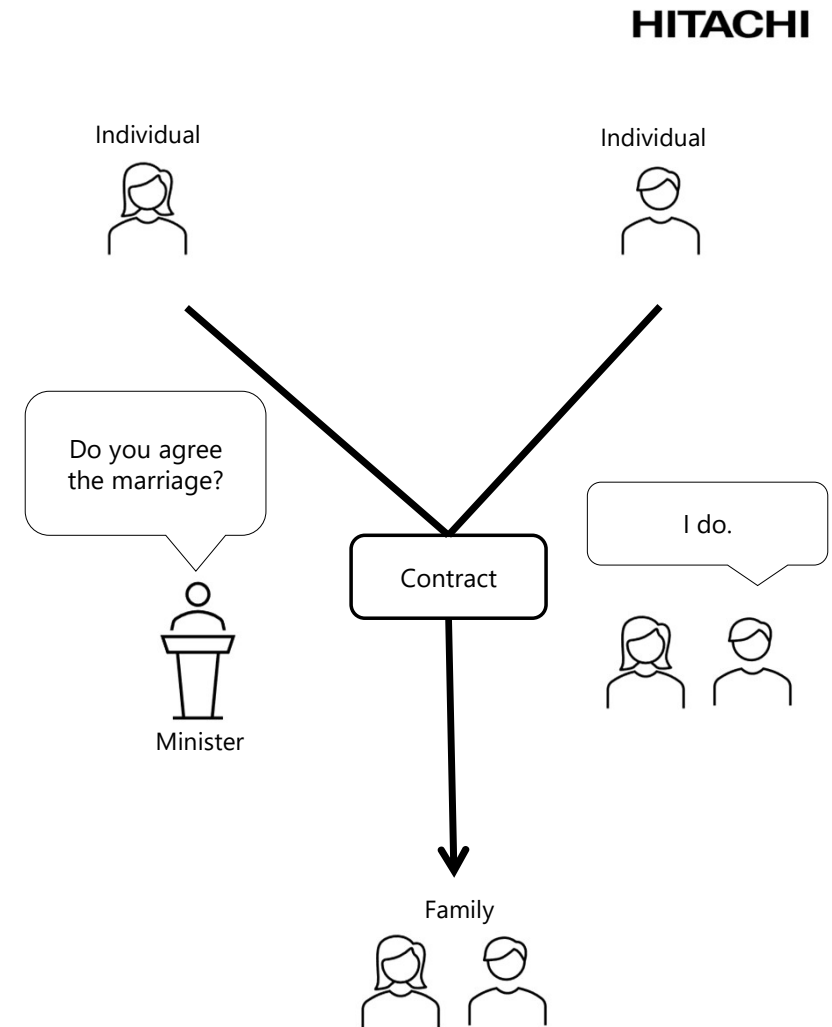


Reflect on the Transaction Concept

The transaction concept comes from contract law.

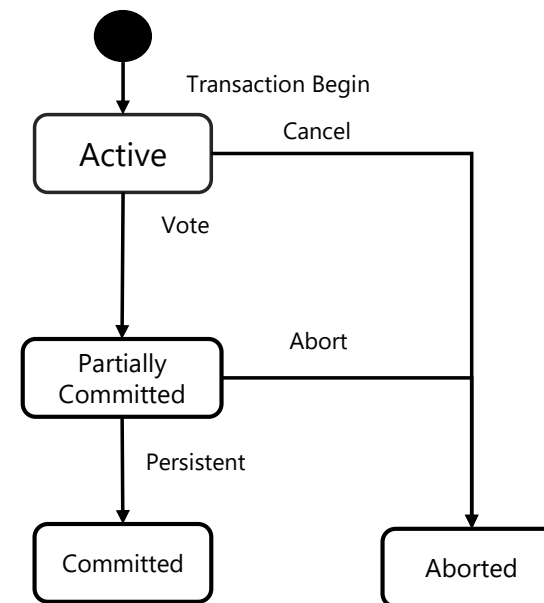
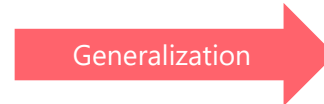
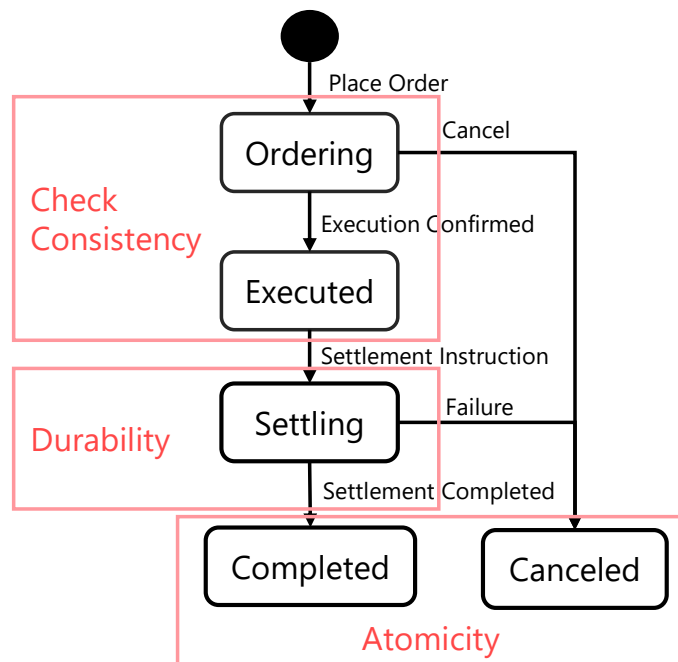
The transaction concept emerges with the following properties:

- the transaction must obey legal protocols. -> **Consistency**
- it either happens or it does not; either all are bound by the contract or none are. -> **Atomicity**
- once a transaction is committed, it cannot be abrogated. -> **Durability**



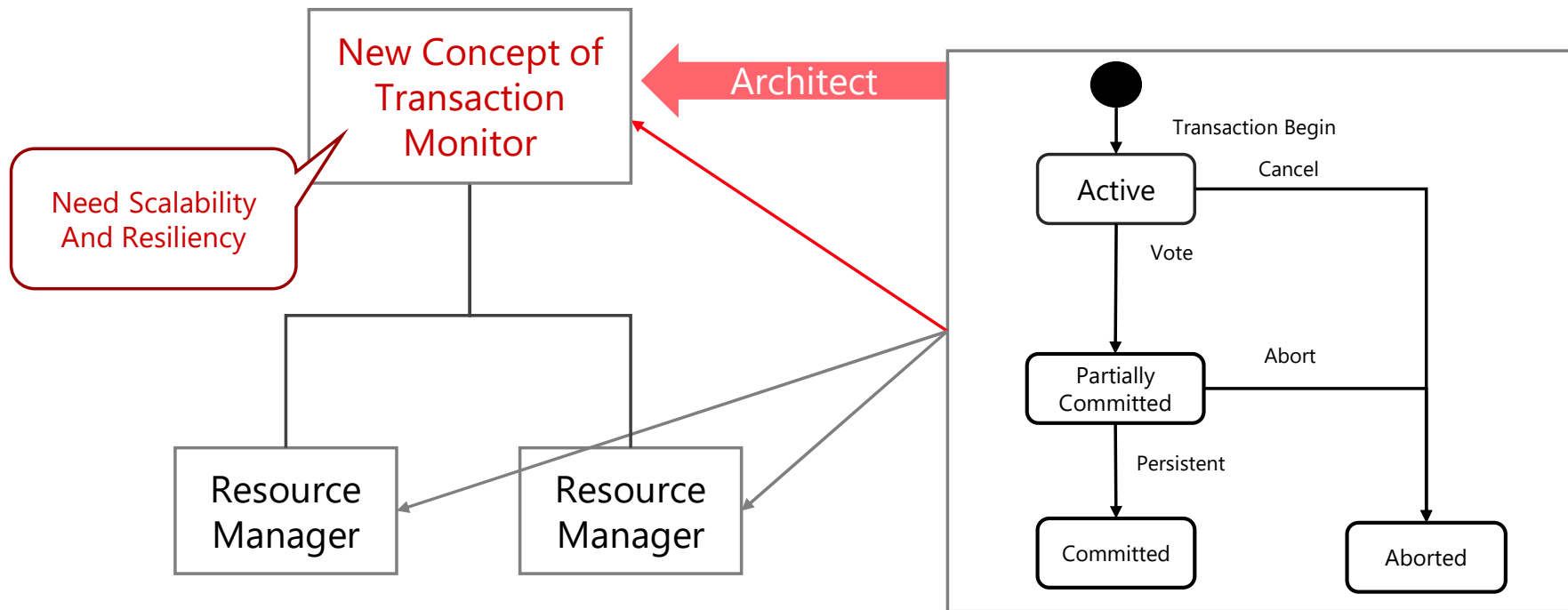
Generalizing the Transaction Concept: ACID and State Transitions

- the transaction must obey legal protocols. -> **Consistency**
- it either happens or it does not; either all are bound by the contract or none are. -> **Atomicity**
- once a transaction is committed, it cannot be abrogated. -> **Durability**



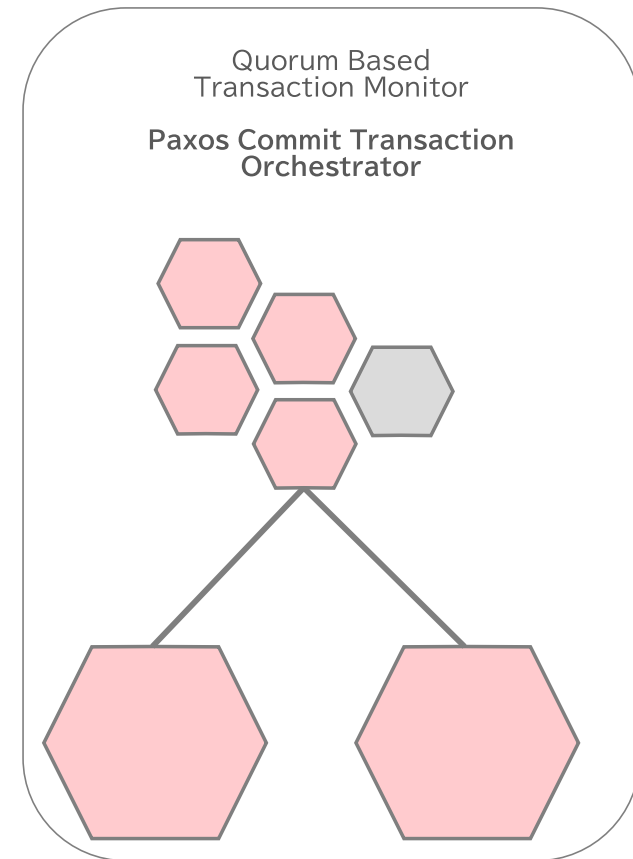
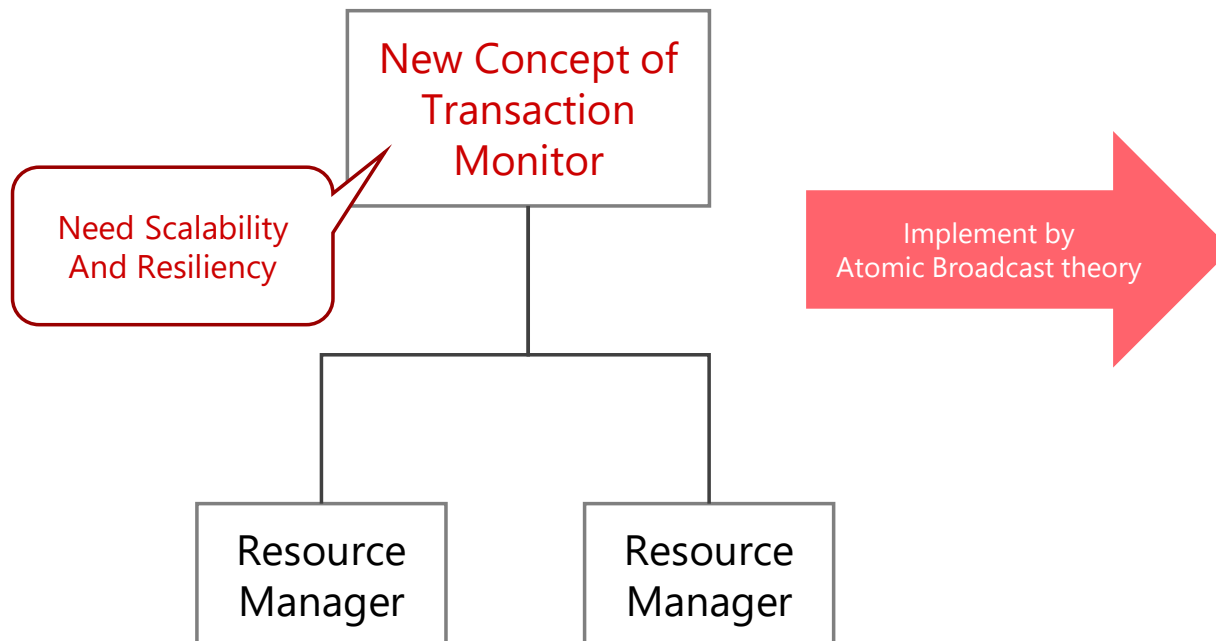
Revisiting Transactions for Cloud-Native Systems: ACID and State Transitions

Let's revisit what a transaction really is. By abstracting it into ACID properties and explicit state transitions, we establish a foundation for building scalable and resilient transaction mechanisms in cloud-native systems.



Transactions Revisited: From ACID to State Transitions

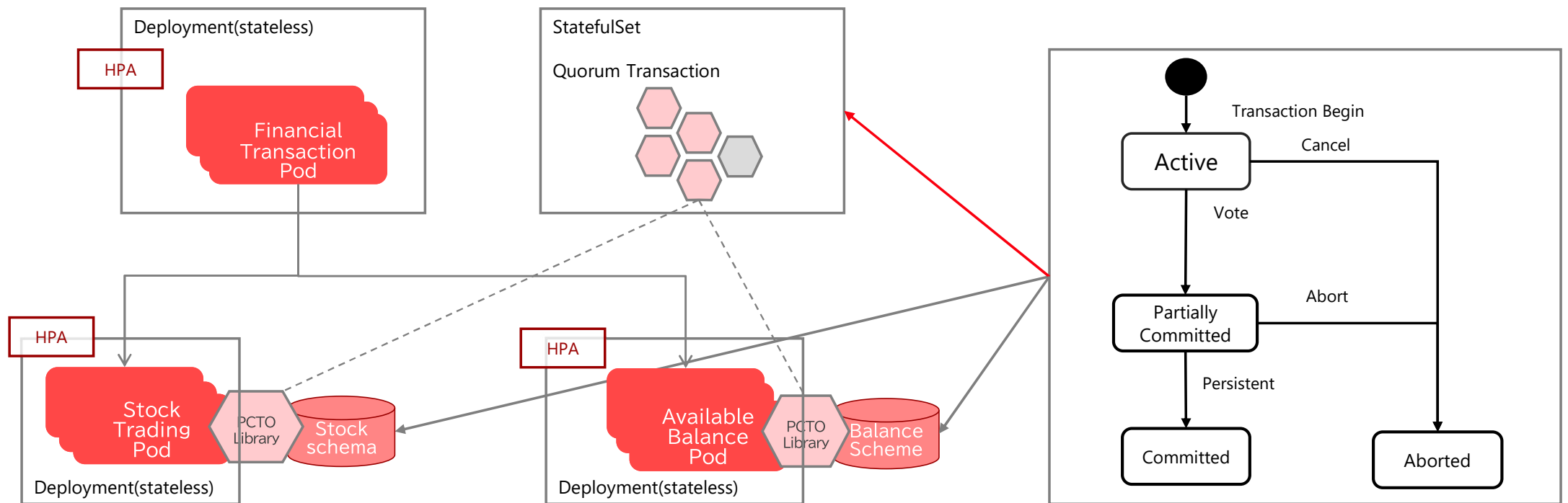
By abstracting transactions into ACID properties and state transitions, we lay the foundation for scalable and resilient distributed systems.



Transaction processing mechanism of Paxos commit
Transaction orchestrator with autoscaling.

Paxos Commit Transaction Orchestrator

Scalability issues in distributed transactions are not solved by tuning consensus, but by re-architecting transaction management itself. Paxos Commit provides that architectural foundation, and Kubernetes makes it practical at scale.

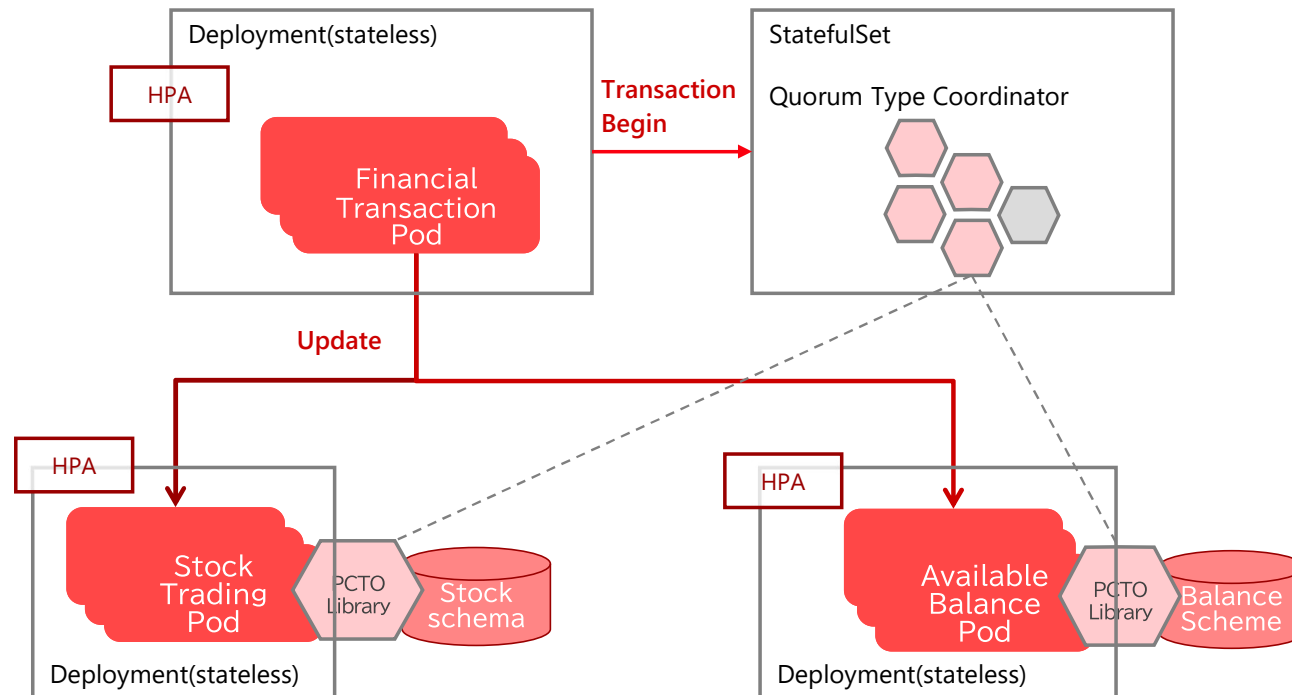


Simplifying Transaction Finality with Paxos Commit

From initiation to completion, the Paxos Commit Transaction Orchestrator proceeds in two steps:

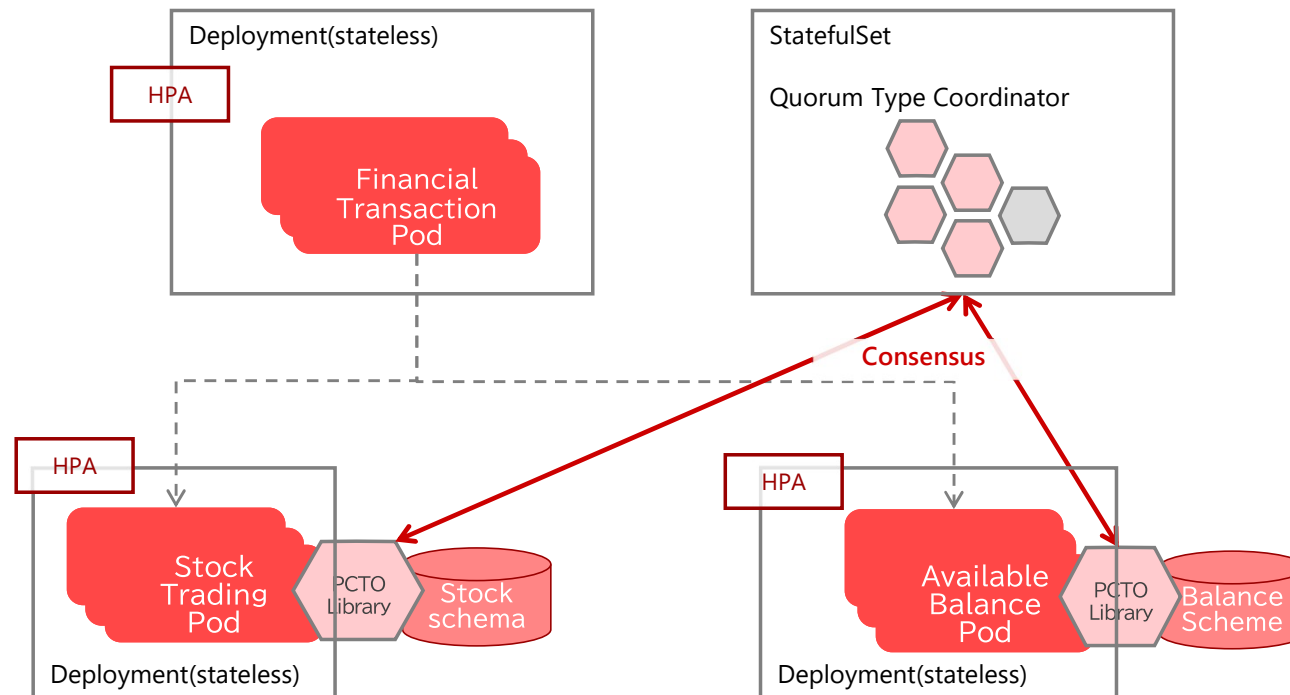
first updating all participating data,

then reaching consensus to either commit or abort the transaction.



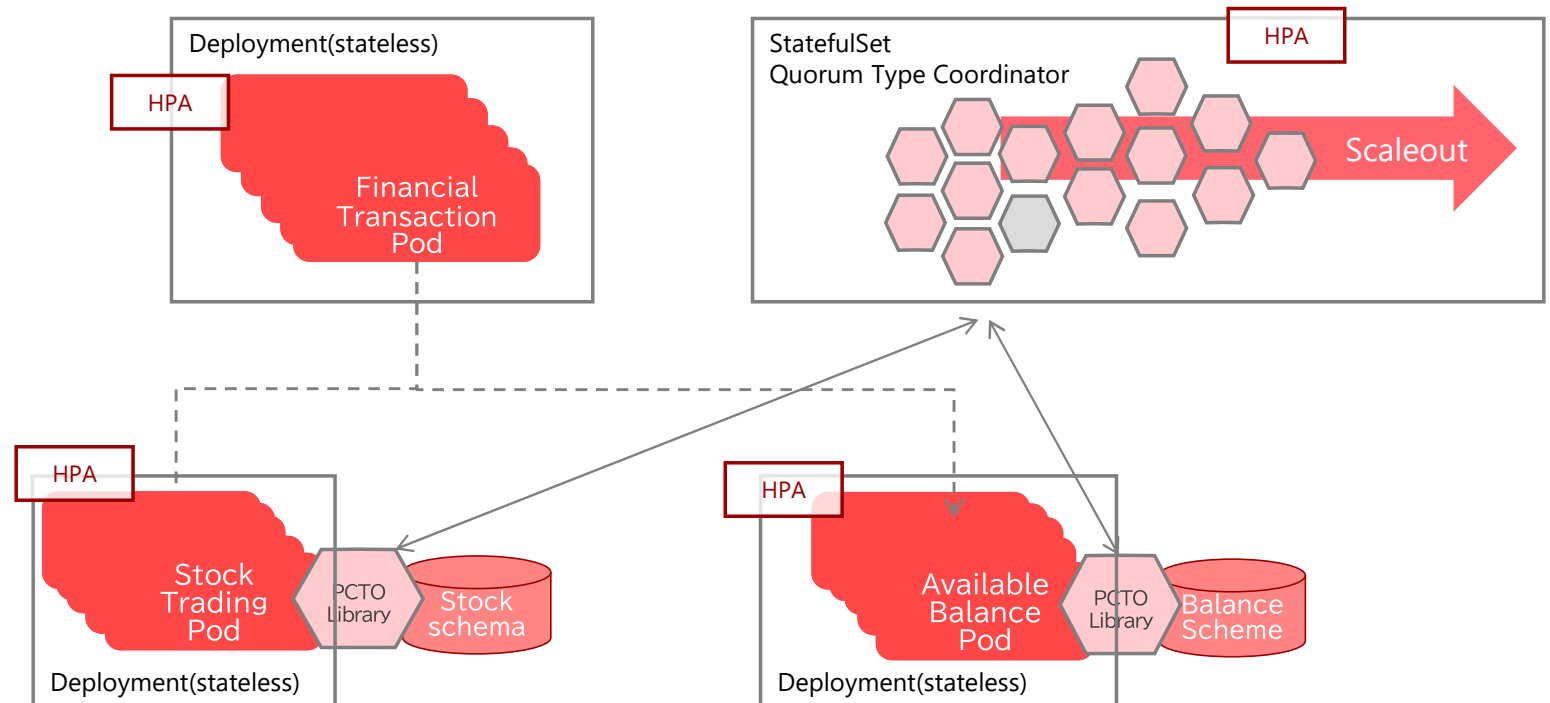
Simplifying Transaction Finality with Paxos Commit

From initiation to completion, the Paxos Commit Transaction Orchestrator proceeds in two steps:
first updating all participating data,
then reaching consensus to either commit or abort the transaction.



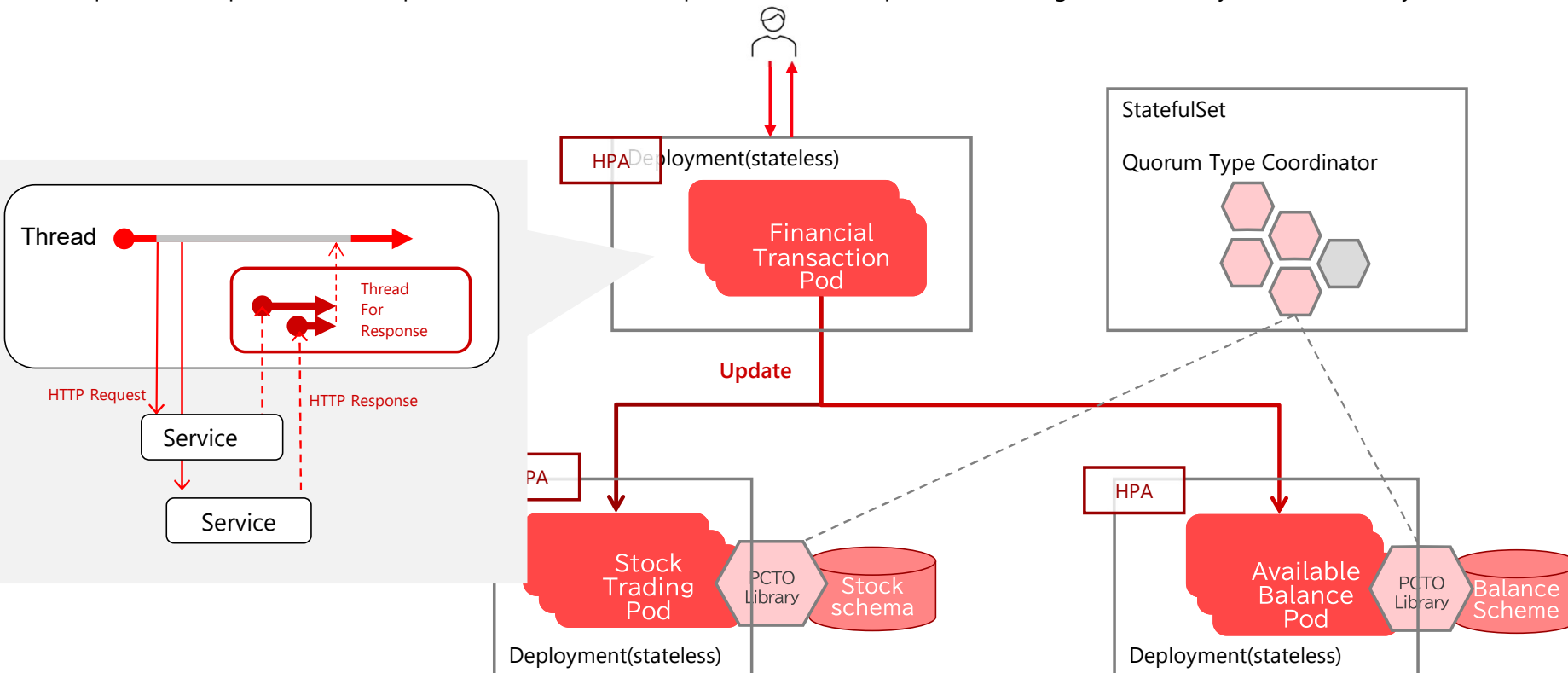
Independent HPA Scaling of Stateless Services and Quorum Coordinators

As transaction volume grows, stateless business applications managed by Deployments scale automatically via HPA, while the Quorum-Type Coordinator managed by a StatefulSet can also be independently auto-scaled using HPA.



Improving Transaction Response Time with Reactive Processing

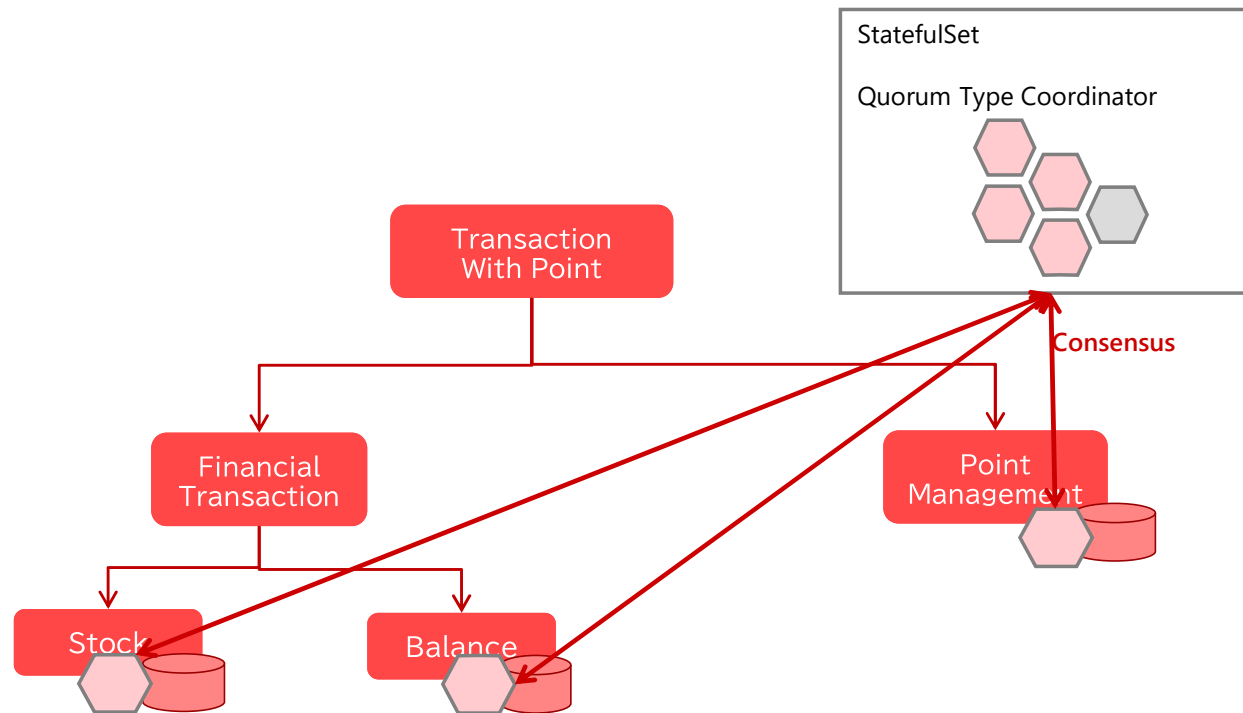
Shorter response times directly improve the user experience of online transactions. By using reactive processing with Spring WebFlux WebClient and separating request and response threads, updates can be executed in parallel without response time being constrained by database latency or microservice scale.



Topology-Independent Transaction Finality with Reactive Processing

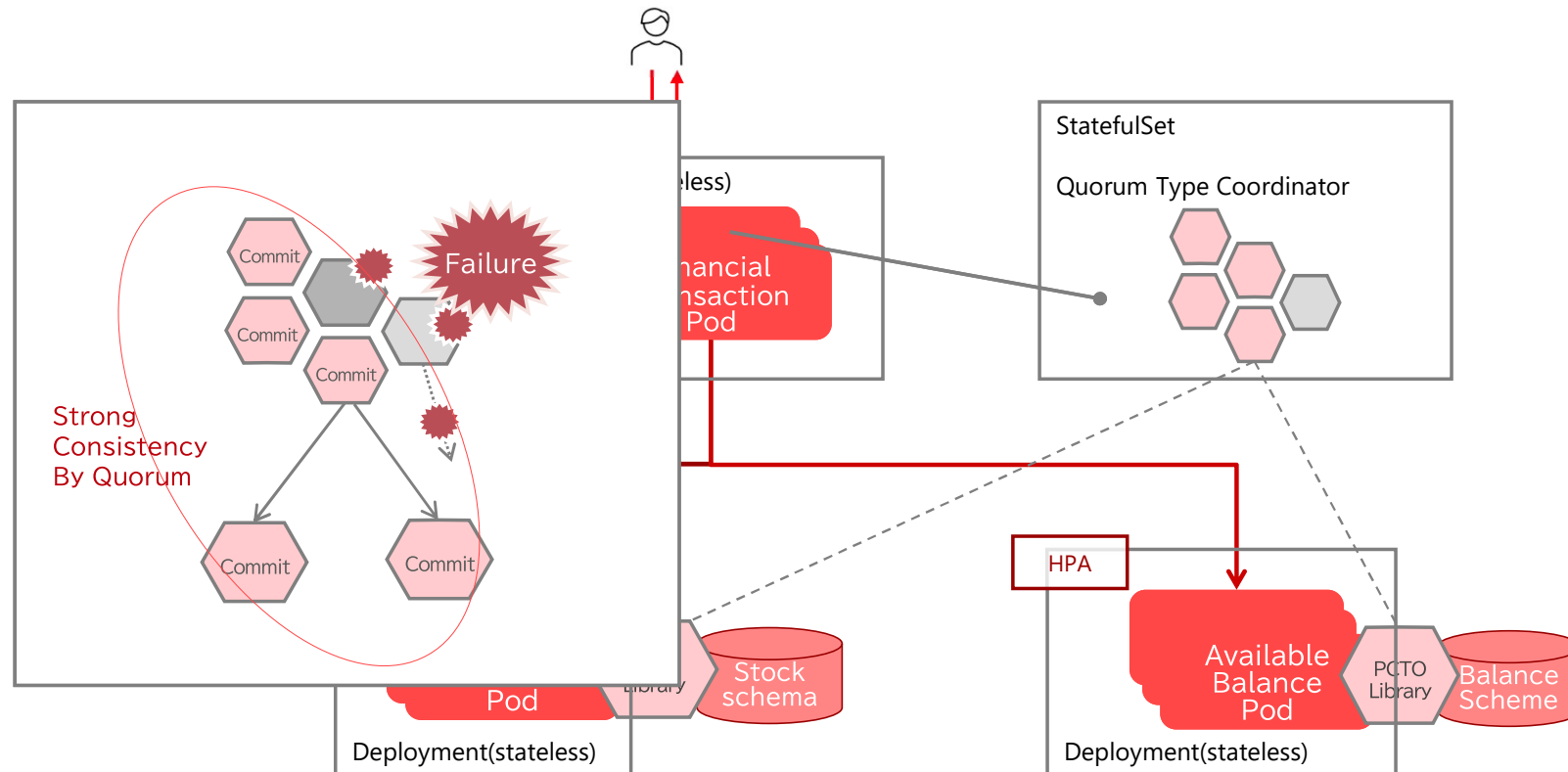
Even as business microservices grow in scale:

1. Reactive processing allows updates to be executed in parallel across multiple components.
2. The transaction resolution path is independent of the microservice topology, and consensus is reached in approximately **0.03 seconds in all cases**.



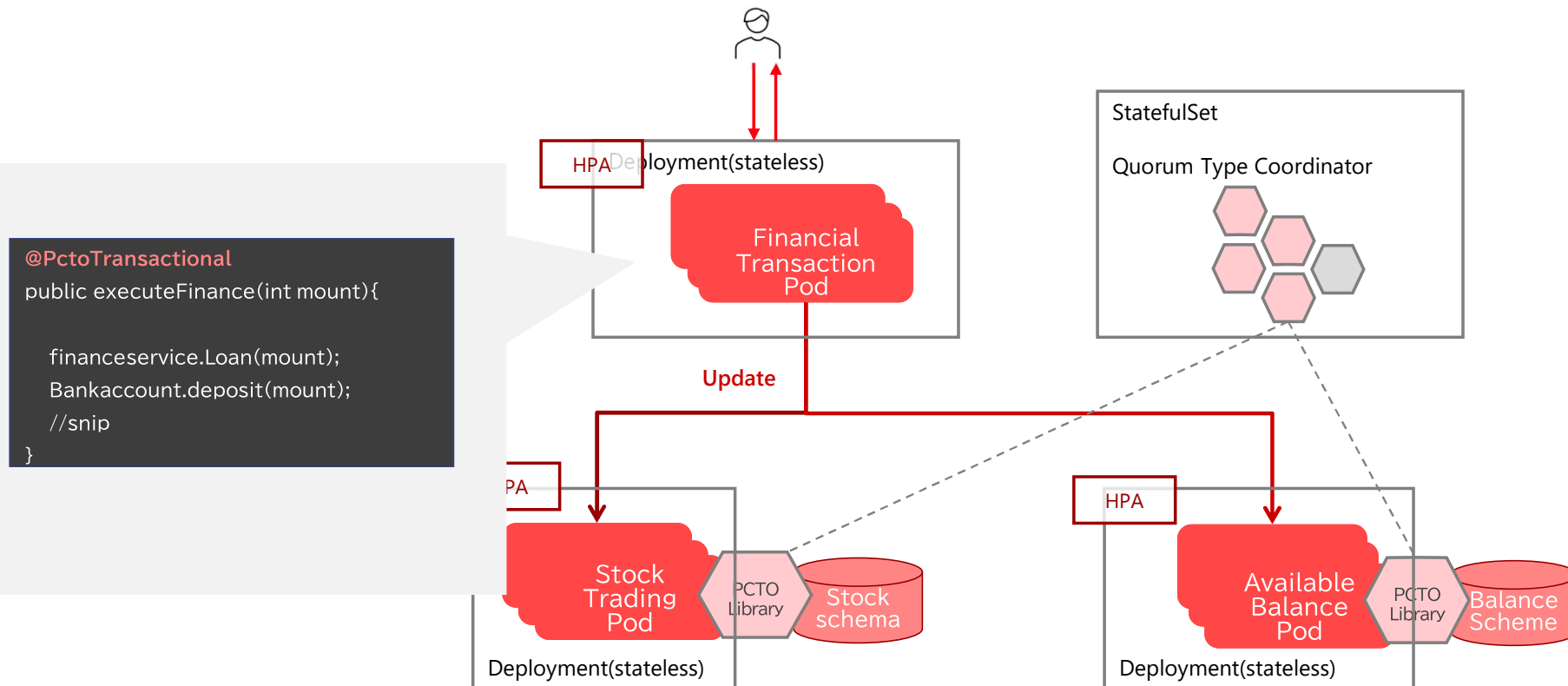
Resiliency

Distributed consensus enables resilient transaction processing: even during partial failures, a surviving majority of Coordinator guarantees consistent results and stable response times of about 0.03 seconds.



Advanced Transactions, Simple Application Code

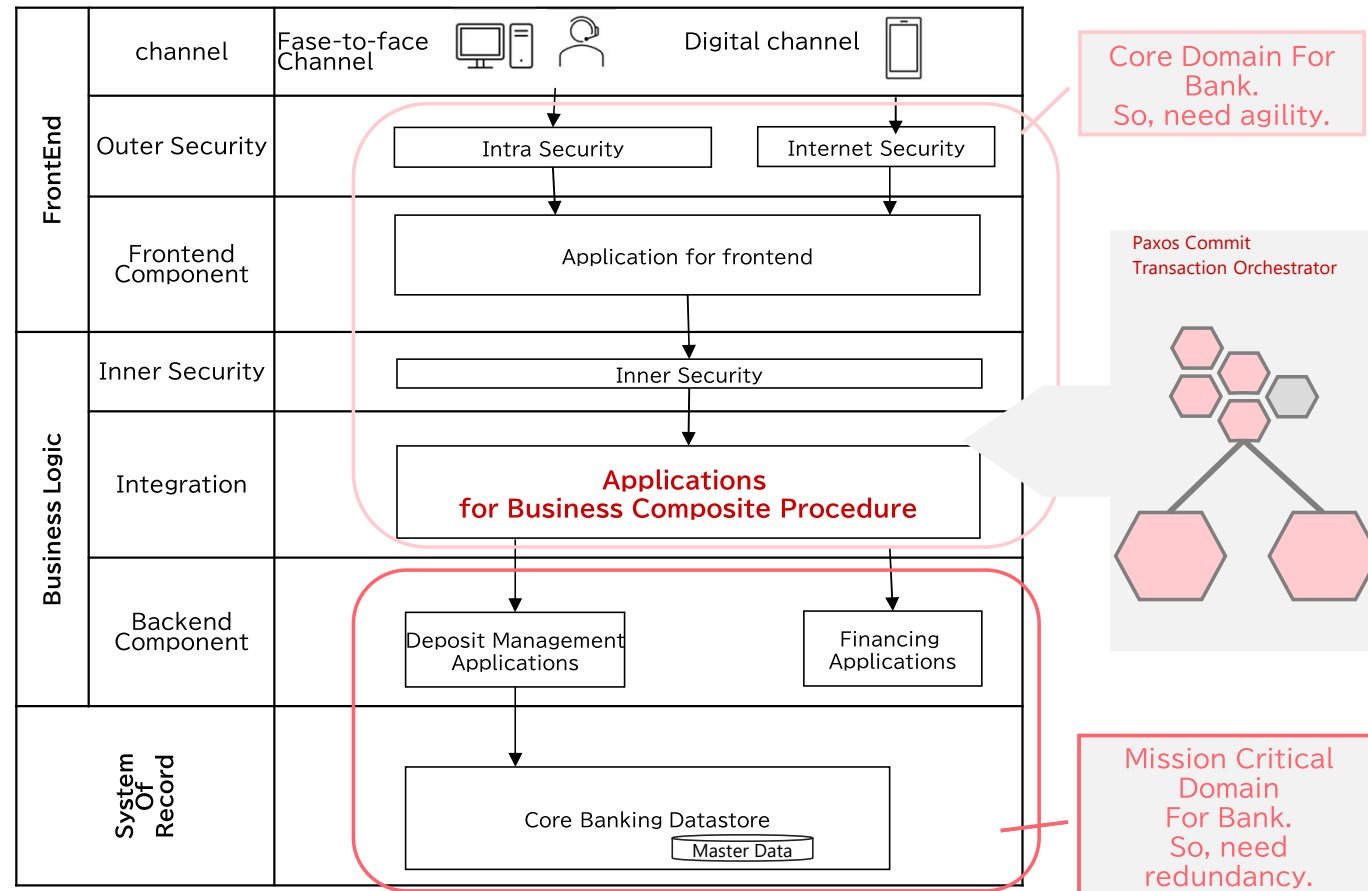
Once we push parallelism, consensus control, and resiliency this far, the next concern is often business-logic pollution: “Will this force extra constraints into my application code?” To avoid that, we offer a Jakarta EE JTA-like API.



Strong Consistency Does Not Have to Limit Scalability

HITACHI

Strong consistency does not have to limit scalability.
The real problem is not Kubernetes,
and it is not Paxos.
It is where consensus is applied.
By rethinking transaction management itself with Paxos
Commit,
we keep business services stateless, autoscalable, and
resilient
— without giving up ACID guarantees.
If your microservices stop scaling,
don't tune consensus harder.
Rethink where consensus belongs.



Contact Us

■Regarding Your Inquiry

Inquiries About This Session

■Contact Information

DataBase Department, Managed & Platform Services Business Division, Hitachi, Ltd.

E-mail: hmp-public-promo@ml.itg.hitachi.co.jp

■Important Information Before Contacting Us

- Personal data we collect will be used only to respond to inquiries concerning this session.
- Personal data collected will be transferred to servers operated by Hitachi, Ltd. located in Japan. Japan has been recognized as providing an adequate level of data protection pursuant to Article 45 of the GDPR. By granting this consent, the data subject hereby acknowledges and expressly agrees to the transfer of their personal data outside the EEA (European Economic Area).
- Before submitting your inquiry, please read “Privacy Policy(*)” and agree to the contents described therein. If you do not agree, we may not be able to respond to your inquiry. Providing personal data in connection with your inquiry is voluntary. However, if such information is not provided, we may not be able to respond to your inquiry.
- Please read “Privacy Policy(*)” carefully, and only if you agree, kindly include the statement below in your email and submit it to the contact address provided above. “I have read the information regarding the protection of personal data and agree to the contents described therein.”
- The data subject may withdraw consent given to Hitachi, Ltd. at any time. To request withdrawal of consent, please contact us using the contact details provided below.
- Personal data obtained will be deleted once we have provided a response to your inquiry, and our company will not retain such personal information thereafter.
- Personal data protection manager for Inquiries : General Manager, DataBase Department, Managed & Platform Services Business Division, Hitachi, Ltd.

(*)「Privacy Policy」: <https://www.hitachi.com/en/privacy/>

Third-Party Trademark Attribution

- etcd and Kubernetes are registered trademarks of The Linux Foundation in the United States and other countries.
- Microsoft, SQL Server, and related product names are trademarks of Microsoft Corporation.
- Oracle and Oracle Database are registered trademarks of Oracle and/or its affiliates.
- PostgreSQL is a trademark of the PostgreSQL Global Development Group.
- Spring and related marks are trademarks of Broadcom and/or its subsidiaries.
- Other names are trademarks or registered trademarks of their respective owners.

HITACHI

Strongly Consistent Transactions with Multiple Resource Types

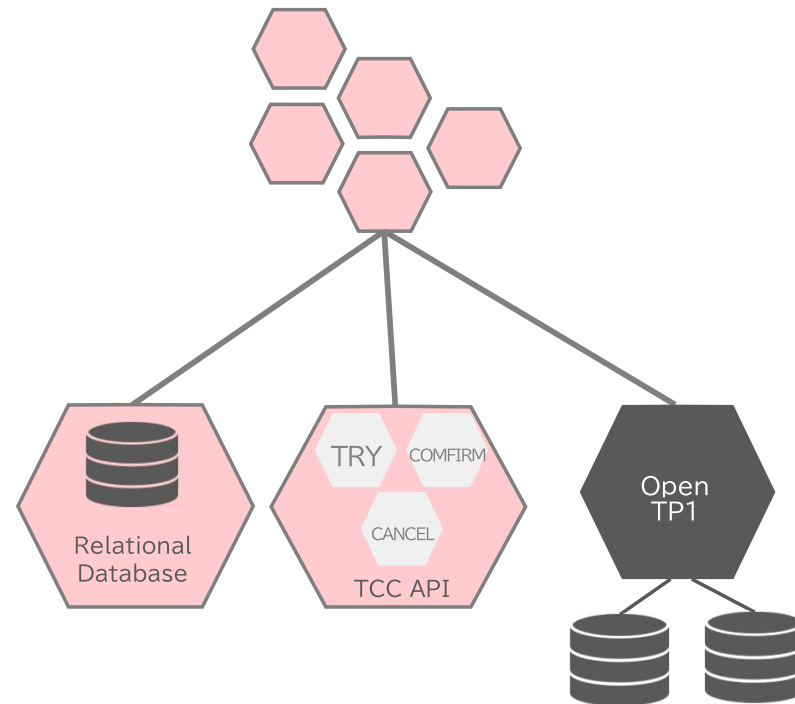
HMP-PCTO allows you to build strongly consistent distributed transaction systems by integrating with multiple types of resources.

The following resources are currently supported.

Relational Database

- PostgreSQL
- SQL Server
- HiRDB
- Oracle DB
- TCC
- Regency Transaction Monitor

Support for additional resources will be expanded incrementally.



CAP Conjecture :

The Conjecture is that distributed systems cannot simultaneously satisfy consistency, availability, and Partition-tolerance all at the same time. Since availability and Partition-tolerance are essential in the cloud, the idea that consistency is ensured by consequence consistency became popular.



- If Partition does not occur, both consistency and availability can be achieved
- Partition rarely occurs in real systems. Even if it does occur, using Paxos to provide redundancy and wait for a commit can prevent the lack of consistency.
- When consistency and Partition-tolerance are required, 100% availability cannot be achieved. However, redundancy can bring availability as close to 100% as possible.

Source : Dwork, Lynch, Stockmeyer, 「Consensus in the presence of partial synchrony」 「Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services」



Using distributed agreement algorithms such as Paxos/Raft, integrity, availability, and fragmentation tolerance can all be satisfied simultaneously