🍳 Virtual Threads Cookbook

> **Java 21+ Virtual Threads with Enterprise Technologies**
> A comprehensive guide for using virtual threads with MongoDB, Cassandra, Kafka, Elasticsearch, Protocol Buffers, and CQRS frameworks.

📋 Table of Contents

Technologies Covered

- **NoSQL DBs**: MongoDB, Cassandra, HBase, Zookeeper
- **Serialization**: Thrift, Protocol Buffers
- **Data Processing**: Hadoop, Kafka
- **Frameworks**: Guice, Spring, Axon (CQRS)
- **Search**: Lucene, Elasticsearch

MongoDB with Virtual Threads

Bulk Message Archiving

```java
@Service
public class MongoVirtualThreadService {
    private final MongoTemplate mongoTemplate;
    private final ExecutorService virtualExecutor;

    @Inject
    public MongoVirtualThreadService(MongoTemplate mongoTemplate) {
        this.mongoTemplate = mongoTemplate;
        this.virtualExecutor = Executors.newVirtualThreadPerTaskExecutor();
    }

    // Bulk message archiving with virtual threads
    public CompletableFuture<List<String>> bulkArchiveMessages(List<ArchivedMessage>
messages) {
        System.out.println("🗄️ Archiving " + messages.size() + " messages with
virtual threads");

        List<CompletableFuture<String>> futures = messages.stream()
            .map(message -> CompletableFuture.supplyAsync(() -> {
                try {
                    // Each message gets its own virtual thread for DB operation
                    mongoTemplate.save(message);

                    // Simulate compliance checking (I/O operation)
                    Thread.sleep(Duration.ofMillis(50));

                    return "Archived: " + message.getId() + " on " +
Thread.currentThread();
                } catch (Exception e) {
                    return "Failed: " + message.getId() + " - " + e.getMessage();
                }
            }, virtualExecutor))
            .toList();

        return CompletableFuture.allOf(futures.toArray(new CompletableFuture[0]))
            .thenApply(v -> futures.stream()
                .map(CompletableFuture::join)
                .toList());
    }
}
```

Parallel Aggregations

```
// Parallel aggregation across multiple collections
public CompletableFuture<Map<String, Object>> parallelAggregations(String userId) {
    // Each aggregation runs in its own virtual thread
    CompletableFuture<Long> messageCount = CompletableFuture.supplyAsync(() -> {
        Query query = new Query(Criteria.where("userId").is(userId));
        return mongoTemplate.count(query, ArchivedMessage.class);
    }, virtualExecutor);

    CompletableFuture<List<TransactionSummary>> transactionSummary =
CompletableFuture.supplyAsync(() -> {
        Aggregation agg = Aggregation.newAggregation(
            Aggregation.match(Criteria.where("userId").is(userId)),
            Aggregation.group("currency").sum("amount").as("total")
        );
        return mongoTemplate.aggregate(agg, "transactions", TransactionSummary.class)
            .getMappedResults();
    }, virtualExecutor);

    // Combine all results
    return CompletableFuture.allOf(messageCount, transactionSummary)
        .thenApply(v -> Map.of(
            "messageCount", messageCount.join(),
            "transactionSummary", transactionSummary.join()
        ));
}
```

Cassandra with Virtual Threads

High-Throughput Message Ingestion

```java
@Service
public class CassandraVirtualThreadService {
    private final CqlSession session;
    private final PreparedStatement insertStatement;

    // High-throughput message ingestion
    public CompletableFuture<String> massiveMessageIngestion(List<UserMessage>
messages) {
        System.out.println("📡 Ingesting " + messages.size() + " messages to
Cassandra");

        try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor())
{

            List<CompletableFuture<Void>> insertFutures = messages.stream()
                .map(message -> CompletableFuture.runAsync(() -> {
                    try {
                        BoundStatement bound = insertStatement.bind(
                            message.getUserId(),
                            message.getTimestamp(),
                            message.getMessageId(),
                            message.getContent()
                        );

                        // Cassandra async operation in virtual thread
                        session.executeAsync(bound)
                            .toCompletableFuture()
                            .join(); // Block this virtual thread until complete

                    } catch (Exception e) {
                        throw new RuntimeException("Failed to insert message: " +
message.getMessageId(), e);
                    }
                }, executor))
                .toList();

            return CompletableFuture.allOf(insertFutures.toArray(new
CompletableFuture[0]))
                .thenApply(v -> "Successfully ingested " + messages.size() + "
messages");
        }
    }
}
```

Parallel Time Range Queries

```
// Parallel data retrieval across time ranges
public CompletableFuture<List<UserMessage>> parallelTimeRangeQuery(
        String userId, List<TimeRange> timeRanges) {

    try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {

        List<CompletableFuture<List<UserMessage>>> rangeFutures = timeRanges.stream()
            .map(range -> CompletableFuture.supplyAsync(() -> {
                BoundStatement bound = selectStatement.bind(
                    userId, range.getStart(), range.getEnd()
                );

                return session.executeAsync(bound)
                    .toCompletableFuture()
                    .join()
                    .map(row -> mapRowToUserMessage(row))
                    .all();
            }, executor))
            .toList();

        return CompletableFuture.allOf(rangeFutures.toArray(new
CompletableFuture[0]))
            .thenApply(v -> rangeFutures.stream()
                .flatMap(future -> future.join().stream())
                .toList());
    }
}
```

Kafka with Virtual Threads

Massive Parallel Publishing

```
@Service
public class KafkaVirtualThreadService {
    private final KafkaTemplate<String, byte[]> kafkaTemplate;
    private final MessageSerializationService serializationService;

    // Massive parallel message publishing
    public CompletableFuture<String>
publishFinancialEvents(List<FinancialTransaction> transactions) {
        System.out.println("📤 Publishing " + transactions.size() + " events to
Kafka");

        try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor())
{

            List<CompletableFuture<SendResult<String, byte[]>>> publishFutures =
transactions.stream()
                .map(transaction -> CompletableFuture.supplyAsync(() -> {
                    try {
                        // Serialize in virtual thread
                        byte[] serializedData =
serializationService.serialize(transaction);

                        ProducerRecord<String, byte[]> record = new ProducerRecord<>(
                            "financial-transactions",
                            transaction.getId(),
                            serializedData
                        );

                        // Kafka send is async, but we wait in this virtual thread
                        return kafkaTemplate.send(record).get();

                    } catch (Exception e) {
                        throw new RuntimeException("Failed to publish: " +
transaction.getId(), e);
                    }
                }, executor))
                .toList();

            return CompletableFuture.allOf(publishFutures.toArray(new
CompletableFuture[0]))
                .thenApply(v -> {
                    long successCount = publishFutures.stream()
                        .mapToLong(future -> future.isCompletedExceptionally() ? 0 :
1)
                        .sum();
```

```
                    return "Published " + successCount + "/" + transactions.size() +
" events";
                });
        }
    }
}
```

Fan-out Processing

```
// Fan-out processing to multiple topics
public CompletableFuture<Void> fanOutProcessing(FinancialTransaction transaction) {
    try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {

        // Send to different topics based on business rules
        List<CompletableFuture<Void>> fanOutTasks = List.of(

            // Compliance topic
            CompletableFuture.runAsync(() -> {
                if (transaction.getRiskScore() > 0.7) {
                    publishToTopic("compliance-alerts", transaction);
                }
            }, executor),

            // Reporting topic
            CompletableFuture.runAsync(() -> {
                publishToTopic("financial-reporting", transaction);
            }, executor),

            // High-value transactions
            CompletableFuture.runAsync(() -> {
                if (transaction.getAmount().compareTo(new BigDecimal("10000")) > 0) {
                    publishToTopic("high-value-transactions", transaction);
                }
            }, executor)
        );

        return CompletableFuture.allOf(fanOutTasks.toArray(new
CompletableFuture[0]));
    }
}
```

Elasticsearch with Virtual Threads

Bulk Document Indexing

```
@Service
public class ElasticsearchVirtualThreadService {
    private final ElasticsearchRestTemplate elasticsearchTemplate;

    // Parallel indexing for massive document sets
    public CompletableFuture<IndexingResult>
bulkIndexDocuments(List<FinancialMessageDocument> documents) {
        System.out.println("🔍 Indexing " + documents.size() + " documents to
Elasticsearch");

        try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor())
{

            // Batch documents for efficient indexing
            int batchSize = 100;
            List<List<FinancialMessageDocument>> batches = partitionList(documents,
batchSize);

            List<CompletableFuture<Integer>> batchFutures = batches.stream()
                .map(batch -> CompletableFuture.supplyAsync(() -> {
                    try {
                        // Each batch gets its own virtual thread
                        elasticsearchTemplate.save(batch);
                        return batch.size();
                    } catch (Exception e) {
                        System.err.println("Batch indexing failed: " +
e.getMessage());
                        return 0;
                    }
                }, executor))
                .toList();

            return CompletableFuture.allOf(batchFutures.toArray(new
CompletableFuture[0]))
                .thenApply(v -> {
                    int totalIndexed = batchFutures.stream()
                        .mapToInt(CompletableFuture::join)
                        .sum();
                    return new IndexingResult(totalIndexed, documents.size());
                });
        }
    }
}
```

Parallel Multi-Index Search

```
// Parallel search across multiple indices
public CompletableFuture<Map<String, SearchResults>> parallelMultiIndexSearch(
        String searchTerm, List<String> indices) {

    try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {

        Map<String, CompletableFuture<SearchResults>> searchFutures =
indices.stream()
            .collect(Collectors.toMap(
                index -> index,
                index -> CompletableFuture.supplyAsync(() -> {
                    try {
                        // Each index search in its own virtual thread
                        Query query = NativeSearchQueryBuilder.newInstance()
                            .withQuery(QueryBuilders.multiMatchQuery(searchTerm,
"content", "subject"))
                            .build();

                        SearchHits<FinancialMessageDocument> hits =
                            elasticsearchTemplate.search(query,
FinancialMessageDocument.class);

                        return new SearchResults(hits.getTotalHits(),
hits.getSearchHits());

                    } catch (Exception e) {
                        return new SearchResults(0, Collections.emptyList());
                    }
                }, executor)
            ));

        return CompletableFuture.allOf(searchFutures.values().toArray(new
CompletableFuture[0]))
            .thenApply(v -> searchFutures.entrySet().stream()
                .collect(Collectors.toMap(
                    Map.Entry::getKey,
                    entry -> entry.getValue().join()
                )));
    }
}
```

Protocol Buffers with Virtual Threads

Parallel Serialization

```
@Service
public class ProtocolBufferVirtualThreadService {

    // Parallel serialization of large datasets
    public CompletableFuture<List<byte[]>>
parallelSerialization(List<FinancialTransaction> transactions) {
        System.out.println("🔄 Serializing " + transactions.size() + " transactions
with Protocol Buffers");

        try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor())
{

            List<CompletableFuture<byte[]>> serializationFutures =
transactions.stream()
                .map(transaction -> CompletableFuture.supplyAsync(() -> {
                    try {
                        // Each serialization in its own virtual thread
                        return TransactionProto.newBuilder()
                            .setTransactionId(transaction.getId())
                            .setUserId(transaction.getUserId())
                            .setAmount(transaction.getAmount().toString())
                            .setCurrency(transaction.getCurrency())
                            .setTimestamp(transaction.getTimestamp().toEpochMilli())
                            .build()
                            .toByteArray();
                    } catch (Exception e) {
                        throw new RuntimeException("Serialization failed for: " +
transaction.getId(), e);
                    }
                }, executor))
                .toList();

            return CompletableFuture.allOf(serializationFutures.toArray(new
CompletableFuture[0]))
                .thenApply(v -> serializationFutures.stream()
                    .map(CompletableFuture::join)
                    .toList());
        }
    }
}
```

Parallel Deserialization and Validation

```
// Parallel deserialization and validation
public CompletableFuture<List<FinancialTransaction>>
parallelDeserialization(List<byte[]> serializedData) {
    try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {

        List<CompletableFuture<FinancialTransaction>> deserializationFutures =
serializedData.stream()
            .map(data -> CompletableFuture.supplyAsync(() -> {
                try {
                    TransactionProto proto = TransactionProto.parseFrom(data);

                    // Validation in virtual thread
                    validateProtoMessage(proto);

                    return FinancialTransaction.builder()
                        .transactionId(proto.getTransactionId())
                        .userId(proto.getUserId())
                        .amount(new BigDecimal(proto.getAmount()))
                        .currency(proto.getCurrency())
                        .timestamp(Instant.ofEpochMilli(proto.getTimestamp()))
                        .build();
                } catch (Exception e) {
                    throw new RuntimeException("Deserialization failed", e);
                }
            }, executor))
            .toList();

        return CompletableFuture.allOf(deserializationFutures.toArray(new
CompletableFuture[0]))
            .thenApply(v -> deserializationFutures.stream()
                .map(CompletableFuture::join)
                .toList());
    }
}
```

CQRS with Axon and Virtual Threads

Parallel Command Processing

```
@Service
public class CQRSVirtualThreadService {
    private final CommandGateway commandGateway;
    private final QueryGateway queryGateway;

    // Parallel command processing
    public CompletableFuture<List<String>>
processTransactionBatch(List<CreateTransactionCommand> commands) {
        System.out.println("⚡ Processing " + commands.size() + " commands with
CQRS");

        try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor())
{

            List<CompletableFuture<String>> commandFutures = commands.stream()
                .map(command -> CompletableFuture.supplyAsync(() -> {
                    try {
                        // Each command in its own virtual thread
                        return commandGateway.sendAndWait(command);
                    } catch (Exception e) {
                        return "Failed: " + command.getTransactionId() + " - " +
e.getMessage();
                    }
                }, executor))
                .toList();

            return CompletableFuture.allOf(commandFutures.toArray(new
CompletableFuture[0]))
                .thenApply(v -> commandFutures.stream()
                    .map(CompletableFuture::join)
                    .toList());
        }
    }
}
```

Parallel Query Execution

```
// Parallel query execution
public CompletableFuture<Map<String, Object>> parallelReportGeneration(String userId)
{
    try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {

        // Multiple queries in parallel virtual threads
        CompletableFuture<List<TransactionView>> transactions =
CompletableFuture.supplyAsync(() -> {
            FindTransactionsByUserQuery query = new
FindTransactionsByUserQuery(userId);
            return queryGateway.query(query,
ResponseTypes.multipleInstancesOf(TransactionView.class)).join();
        }, executor);

        CompletableFuture<UserProfileView> userProfile =
CompletableFuture.supplyAsync(() -> {
            FindUserProfileQuery query = new FindUserProfileQuery(userId);
            return queryGateway.query(query,
ResponseTypes.instanceOf(UserProfileView.class)).join();
        }, executor);

        CompletableFuture<List<ComplianceAlert>> alerts =
CompletableFuture.supplyAsync(() -> {
            FindComplianceAlertsQuery query = new FindComplianceAlertsQuery(userId);
            return queryGateway.query(query,
ResponseTypes.multipleInstancesOf(ComplianceAlert.class)).join();
        }, executor);

        return CompletableFuture.allOf(transactions, userProfile, alerts)
            .thenApply(v -> Map.of(
                "transactions", transactions.join(),
                "userProfile", userProfile.join(),
                "complianceAlerts", alerts.join()
            ));
    }
}
```

Comprehensive Demo

```
@Component
public class VirtualThreadCookbookDemo {

    public void runComprehensiveDemo() {
        System.out.println("🍳 Virtual Threads Cookbook -  Tech Stack Demo");

System.out.println("============================================================");

        // Simulate massive concurrent processing
        try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor())
{

            Instant start = Instant.now();

            // Create sample data
            List<FinancialTransaction> transactions =
createSampleTransactions(10_000);
            List<ArchivedMessage> messages = createSampleMessages(5_000);

            // Parallel processing across all technologies
            CompletableFuture<String> mongoArchiving =
CompletableFuture.supplyAsync(() -> {
                System.out.println("📁 MongoDB: Archiving messages...");
                return "MongoDB: Archived " + messages.size() + " messages";
            }, executor);

            CompletableFuture<String> cassandraIngestion =
CompletableFuture.supplyAsync(() -> {
                System.out.println("🗃️ Cassandra: Ingesting time-series data...");
                return "Cassandra: Ingested " + transactions.size() + "
transactions";
            }, executor);

            CompletableFuture<String> kafkaPublishing =
CompletableFuture.supplyAsync(() -> {
                System.out.println("📡 Kafka: Publishing events...");
                return "Kafka: Published " + transactions.size() + " events";
            }, executor);

            CompletableFuture<String> elasticsearchIndexing =
CompletableFuture.supplyAsync(() -> {
                System.out.println("🔍 Elasticsearch: Indexing documents...");
                return "Elasticsearch: Indexed " + messages.size() + " documents";
            }, executor);
```

```
            // Wait for all operations to complete
            CompletableFuture.allOf(
                mongoArchiving, cassandraIngestion, kafkaPublishing,
elasticsearchIndexing
            ).join();

            Duration elapsed = Duration.between(start, Instant.now());

            System.out.println("\n✅ All operations completed in " +
elapsed.toMillis() + "ms");
            System.out.println("🚀 Virtual threads enabled massive parallel
processing!");
        }
    }
}
```

Key Benefits

🚀 Massive Scalability

- **Handle 10,000+ concurrent operations** with minimal memory footprint
- **Perfect for I/O-heavy workloads** common in financial services
- **No thread pool exhaustion** with database connections

⚡ Real-world Performance Improvements

- **Message Archiving**: Process 100,000 emails/chats simultaneously
- **Compliance Scanning**: Parallel analysis across millions of documents
- **Risk Assessment**: Concurrent processing of financial transactions
- **eDiscovery**: Search across terabytes of data in parallel

🎯 Best Practices

✅ Do's

- **Use try-with-resources** for ExecutorService auto-cleanup
- **Leverage virtual threads for I/O-intensive operations**
- **Combine with async APIs** (Cassandra, MongoDB async drivers)
- **Use for high-concurrency scenarios** (10,000+ operations)
- **Structure code with CompletableFuture** for composability

❌ Don'ts

- **Avoid for CPU-intensive tasks** (use ForkJoinPool instead)
- **Don't use excessive synchronized blocks** (can pin virtual threads)
- **Avoid blocking on virtual threads unnecessarily**
- **Don't create virtual threads for short-lived operations**

💡 Performance Tips

- **Batch operations** when possible (Elasticsearch bulk indexing)
- **Use appropriate timeouts** for I/O operations
- **Monitor virtual thread metrics** in production
- **Consider using virtual thread factories** for custom naming

Conclusion

Virtual threads transform enterprise Java applications from being **limited by thread pool sizes** to being **limited only by business logic and I/O capacity**. For 's technology stack, this means:

- **Unprecedented scalability** for message archiving and compliance systems
- **Simplified concurrency models** without complex thread pool management
- **Better resource utilization** across MongoDB, Cassandra, Kafka, and Elasticsearch
- **Enhanced system responsiveness** under high load conditions

This cookbook demonstrates how virtual threads enable massive parallel processing that would be impossible with traditional platform threads, making them perfect for the high-throughput, I/O-intensive workloads typical in financial services technology.