

**Prompt Engineering Playbook**



**Confidentiality**

“This document is the property of Value

Momentum Software Services Private Ltd.”

**Contents**

**Basic Prompts**

**Advanced Prompt Techniques**

**Domain Specific Prompts**

**Troubleshooting Prompts**

**Spring boot Prompts**

**Apache Kafka Prompts**

**SQL & Database Prompts**

**Redis Prompts**

**Design Patterns Prompts**

**Java 8 17 21 Prompts**

**JPA & Hibernate Prompts**

**Basic Prompt Patterns**

**1. Code Completion**

// Calculate the area of a circle given radius

function calculateArea(radius) {

**2. Function Generation**

// Write a Python function to validate email addresses using regex

**3. Code Explanation**

# Explain what this Script code does:

const result = array.reduce((acc, val) => acc + val, 0);

**4. Code Translation**

// Convert this Python code to TypeScript:

def greet(name):

return f"Hello, {name}"

**Advanced Prompt Techniques**

**1. Step-by-Step Implementation**

/\*

Create a React component for a login form with:

1. Email and password fields

2. Form validation

3. Submit handler

4. Error messages

\*/

**2. Constrained Solutions**

// Write a SQL query to find duplicate records in a users table

// Use window functions, no self-joins

**3. Test Case Generation**

// Generate Jest test cases for this sorting function

function sortArray(arr) {

return arr.sort((a, b) => a - b);

}

**4. Documentation**

"""

Generate docstring for this function in Google style format:

def calculate\_tax(income, deductions):

taxable = income - deductions

if taxable > 0:

return taxable \* 0.2

return 0

"""

**Domain-Specific Prompts**

**Web Development**

// Create an Express.js middleware to verify JWT tokens

// with error handling for expired and invalid tokens

**Data Science**

# Pandas code to clean a dataset by:

# 1. Removing rows with missing values

# 2. Converting date column to datetime

# 3. Standardizing text columns to lowercase

**System Programming**

// Write a Java method that spawns 10 worker threads

// to process items from a channel with graceful shutdown

**DevOps**

# Write a Dockerfile for a Node.js 18 application

# with multi-stage build and production optimizations

**Troubleshooting Prompts**

**1. Error Diagnosis**

"""

I'm getting this error when running my Python code:

"ValueError: invalid literal for int() with base 10: 'abc'"

How can I fix it?

"""

**2. Performance Optimization**

// Optimize this Script function for better runtime:

function findDuplicates(arr) {

let result = [];

for (let i = 0; i < arr.length; i++) {

for (let j = i + 1; j < arr.length; j++) {

if (arr[i] === arr[j]) result.push(arr[i]);

}

}

return result;

}

**3. Security Review**

"""

Review this authentication code for security vulnerabilities:

def authenticate(username, password):

user = User.query.filter\_by(username=username).first()

if user and user.password == password:

return True

return False

"""

**Best Practices**

1. **Start small** - Begin with simple prompts and gradually add complexity
2. **Use comments** - Prefix prompts with // or # for better recognition
3. **Provide examples** - Show sample input/output when possible
4. **Break down problems** - Divide complex tasks into smaller prompts
5. **Review outputs** - Always validate Copilot's suggestions

**Anti-Patterns to Avoid**

* Vague prompts: "Write some code"
* Overly complex single prompts
* Assuming all outputs are correct without review
* Ignoring security implications of generated code

**Example Workflow**

1. Define the task in a comment
2. Let Copilot generate initial code
3. Review and refine the output
4. Add specific constraints if needed
5. Request tests or documentation
6. Iterate until satisfied

Remember that Copilot is a tool to augment your productivity, not replace your expertise. Always apply your professional judgment to its outputs.

**1. Spring Boot Prompts**

**A. REST API Development**

/\*\*

\* Create a Spring Boot REST controller for a User Management System with:

\* 1. GET /users - Fetch all users (paginated)

\* 2. GET /users/{id} - Fetch user by ID

\* 3. POST /users - Create a new user (with validation)

\* 4. PUT /users/{id} - Update user details

\* 5. DELETE /users/{id} - Delete a user

\*

\* Use:

\* - ResponseEntity for responses

\* - DTOs for request/response

\* - @Valid for validation

\* - Global exception handling

\*/

**B. Spring Security & JWT**

/\*\*

\* Implement JWT-based authentication in Spring Boot with:

\* 1. User registration (/auth/register)

\* 2. User login (/auth/login) returning JWT

\* 3. Role-based authorization (ADMIN, USER)

\* 4. Secure endpoints using @PreAuthorize

\* 5. Token refresh mechanism

\*

\* Use:

\* - Spring Security 6

\* - JJWT library

\* - Password encryption (BCrypt)

\*/

**C. Spring Data JPA & Transactions**

/\*\*

\* Create a Spring Data JPA repository with:

\* 1. Custom query to find users by email and status

\* 2. @Transactional method to transfer money between accounts (optimistic locking)

\* 3. Auditing (createdAt, updatedAt)

\* 4. Soft delete using @Where clause

\*/

**D. Caching with Spring Cache**

/\*\*

\* Implement caching in a Spring Boot service layer with:

\* 1. @Cacheable on a method fetching product by ID

\* 2. @CacheEvict when updating/deleting a product

\* 3. TTL of 5 minutes

\* 4. Conditional caching (only cache if product price > 100)

\*/

**2. Apache Kafka Prompts**

**A. Kafka Producer & Consumer**

/\*\*

\* Create a Kafka producer in Spring Boot that:

\* 1. Sends order events to "orders-topic" in JSON format

\* 2. Uses KafkaTemplate with retries (3 attempts)

\* 3. Implements a custom serializer

\* 4. Handles Producer errors with a callback

\*/

/\*\*

\* Create a Kafka consumer in Spring Boot that:

\* 1. Listens to "orders-topic" with a consumer group

\* 2. Deserializes JSON into OrderDTO

\* 3. Implements manual acknowledgment

\* 4. Handles duplicate messages (idempotent processing)

\*/

**B. Kafka Streams & Event Processing**

/\*\*

\* Implement a Kafka Streams application that:

\* 1. Reads from "transactions-topic"

\* 2. Filters transactions with amount > $1000

\* 3. Aggregates total transactions per user

\* 4. Writes results to "high-value-transactions-topic"

\*/

**C. Dead Letter Queue (DLQ) Handling**

/\*\*

\* Implement a Kafka DLQ mechanism where:

\* 1. Failed messages go to "orders-dlq-topic"

\* 2. Retry 3 times before sending to DLQ

\* 3. Logs DLQ events for manual recovery

\*/

**3. SQL & Database Optimization**

**A. Complex SQL Queries**

sql

-- Write a SQL query to:

-- 1. Find the top 5 customers by total order amount

-- 2. Include their name, email, and total spent

-- 3. Only consider completed orders in the last 6 months

-- 4. Use window functions for ranking

**B. Indexing & Query Optimization**

sql

-- Optimize this slow-running query (provide execution plan):

SELECT \* FROM orders

WHERE customer\_id IN (SELECT id FROM customers WHERE status = 'ACTIVE')

ORDER BY created\_at DESC

LIMIT 100;

**C. Database Transactions & Locking**

sql

-- Write a transaction that:

-- 1. Deducts balance from Account A

-- 2. Adds balance to Account B

-- 3. Uses row-level locking to prevent race conditions

-- 4. Rolls back on failure

**4. Redis Prompts**

**A. Caching with Redis**

/\*\*

\* Implement Redis caching in Spring Boot for:

\* 1. Product catalog (cache-aside pattern)

\* 2. TTL of 1 hour

\* 3. Cache eviction on updates

\* 4. Fallback to DB on cache miss

\*/

**B. Rate Limiting with Redis**

/\*\*

\* Implement API rate limiting (10 requests/minute) using:

\* 1. Redis + Lua script for atomic increments

\* 2. Sliding window algorithm

\* 3. Returns "429 Too Many Requests" on exceed

\*/

**C. Distributed Locking**

/\*\*

\* Implement a distributed lock using Redis to:

\* 1. Prevent duplicate order processing

\* 2. Auto-release lock after 30 seconds

\* 3. Retry 3 times if lock is held

\*/

**5. Design Patterns**

**A. Singleton (Thread-Safe)**

/\*\*

\* Implement a thread-safe Singleton in using:

\* 1. Double-checked locking

\* 2. volatile keyword

\* 3. Private constructor

\*/

**B. Factory Pattern**

/\*\*

\* Implement a PaymentProcessorFactory that:

\* 1. Returns CreditCardProcessor or PayPalProcessor based on input

\* 2. Follows Open/Closed Principle

\*/

**C. Strategy Pattern**

/\*\*

\* Implement a sorting utility using Strategy Pattern with:

\* 1. BubbleSort, QuickSort, and MergeSort strategies

\* 2. Dynamic switching at runtime

\*/

**D. Circuit Breaker (Resilience4j)**

/\*\*

\* Implement a Circuit Breaker for an external API call:

\* 1. Opens after 5 failures in 10 seconds

\* 2. Half-Open state after 30 seconds

\* 3. Fallback method when circuit is open

\*/

**Tips for Effective Copilot Usage**

**Be explicit** – Specify frameworks, libraries, and constraints.  
**Use step-by-step breakdowns** – Copilot works better with structured prompts.  
**Include error handling** – Ask for retries, fallbacks, and logging.  
**Request optimizations** – Ask for performance, security, or scalability improvements.

By using these prompts, you can **generate high-quality, production-grade code** with GitHub Copilot efficiently! 🚀

github copilot prompts for 8 17 21

**GitHub Copilot Prompts for 8, 17, and 21 Features**

This guide provides **optimized prompts** to generate code using  **8, 17, and 21** features with GitHub Copilot.

**1. 8 Features (Lambdas, Streams, Optional, etc.)**

**A. Lambda Expressions**

// Convert this anonymous Runnable to a lambda expression

new Thread(new Runnable() {

@Override

public void run() {

System.out.println("Running in a thread");

}

}).start();

**B. Stream API (Filter, Map, Reduce, Collectors)**

// Given a list of integers, use Streams to:

// 1. Filter even numbers

// 2. Square each number

// 3. Sum all squared values

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);

**C. Optional for Null Safety**

// Refactor this null-checking code using Optional

String name = getNameFromDB();

if (name != null) {

System.out.println(name.toUpperCase());

} else {

System.out.println("Name not found");

}

**D. Method References**

// Replace this lambda with a method reference

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

names.forEach(s -> System.out.println(s));

**E. Default & Static Methods in Interfaces**

// Create an interface Logger with:

// 1. A default method log(String message)

// 2. A static method getCurrentTimestamp()

**2. 17 Features (Sealed Classes, Pattern Matching, Records, etc.)**

**A. Sealed Classes & Interfaces**

// Define a sealed hierarchy for shapes:

// 1. Sealed interface Shape permits Circle, Rectangle

// 2. Record Circle(double radius) implements Shape

// 3. Record Rectangle(double width, double height) implements Shape

**B. Pattern Matching for instanceof**

// Refactor this using 17 pattern matching

if (obj instanceof String) {

String s = (String) obj;

System.out.println(s.length());

}

**C. Records (Immutable Data Classes)**

// Convert this class to a Record

public final class Person {

private final String name;

private final int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// Getters, equals, hashCode, toString

}

**D. Text Blocks (Multi-line Strings)**

// Refactor this SQL query using 17 text blocks

String sql = "SELECT id, name, email\n" +

"FROM users\n" +

"WHERE status = 'ACTIVE'";

**E. Switch Expressions (Arrow Syntax)**

// Rewrite this switch statement using 17 switch expressions

String day = "MONDAY";

switch (day) {

case "MONDAY":

case "FRIDAY":

System.out.println("Weekday");

break;

case "SATURDAY":

case "SUNDAY":

System.out.println("Weekend");

break;

default:

System.out.println("Invalid day");

}

**3. 21 Features (Virtual Threads, Sequenced Collections, etc.)**

**A. Virtual Threads (Project Loom)**

// Create 1000 virtual threads to simulate concurrent tasks

// Each thread should print its number and sleep for 1 second

**B. Sequenced Collections (New Collection Methods)**

// Given a LinkedHashSet, use 21 methods to:

// 1. Get the first element

// 2. Get the last element

// 3. Remove the first element

Set<String> orderedSet = new LinkedHashSet<>(List.of("A", "B", "C"));

**C. String Templates (PREVIEW in 21)**

// Use 21 string templates to format this JSON

String name = "Alice";

int age = 30;

String json = "{\n" +

" \"name\": \"" + name + "\",\n" +

" \"age\": " + age + "\n" +

"}";

**D. Pattern Matching for Switch (Finalized in 21)**

// Use pattern matching in switch to handle different shapes

Shape shape = new Circle(5.0);

switch (shape) {

case Circle c -> System.out.println("Circle with radius: " + c.radius());

case Rectangle r -> System.out.println("Rectangle area: " + (r.width() \* r.height()));

default -> System.out.println("Unknown shape");

}

**E. Structured Concurrency (PREVIEW in 21)**

// Use StructuredTaskScope to fetch user data and orders concurrently

// If either fails, cancel the other

**4. Best Practices for + Copilot Prompts**

* **Specify version** – "Use 17 records" or " 8 streams"
* **Ask for modern alternatives** – "Rewrite using 21 virtual threads"
* **Request optimizations** – "Make this code thread-safe"
* **Include error handling** – "Add null checks using Optional"

**GitHub Copilot Prompts for : Interfaces, Annotations, Collections, Streams & File Handling**

**1. Interface Prompts**

**Basic Interface Definition**

// Create a Drawable interface with:

// - A default method draw() that prints "Drawing shape"

// - An abstract method resize(int factor)

**Functional Interface**

// Create a @FunctionalInterface called StringProcessor with:

// - A single abstract method process(String input)

// - A default method toUpperCase() that processes the string

**Interface Inheritance**

// Create interface Auditable extending AutoCloseable with:

// - An abstract method audit()

// - Inherited close() behavior with custom implementation

**2. Annotation Prompts**

**Custom Annotation**

// Create a @Timed annotation that:

// - Can be applied only to methods

// - Records method execution time

// - Has optional 'threshold' parameter (default 100ms)

**Annotation Processing**

// Write code to check if a method has @Deprecated annotation

// and print a warning message if found

**Meta-Annotations**

// Create a @SecureAPI annotation that:

// - Is itself annotated with @Target(METHOD)

// - Has retention policy RUNTIME

// - Contains required 'roles' array parameter

**3. Collection Class Prompts**

**List Operations**

// Given List<String>, write code to:

// 1. Remove duplicates

// 2. Sort alphabetically

// 3. Convert to uppercase

List<String> names = Arrays.asList("John", "Alice", "Bob", "Alice");

**Map Operations**

// Create a Map<String, Integer> from List<String> where:

// - Key is the string

// - Value is string length

// - Exclude strings longer than 10 chars

**Custom Comparator**

// Sort List<Employee> by:

// 1. Department (ascending)

// 2. Salary (descending)

// 3. Name (ascending)

**4. Stream API Prompts**

**Basic Stream Operations**

// Given List<Integer>, use streams to:

// 1. Filter even numbers

// 2. Square remaining numbers

// 3. Calculate average

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

**Advanced Stream Operations**

// Partition List<Person> into two maps:

// 1. Adults (age >= 18) by name

// 2. Minors (age < 18) by age group

**Parallel Streams**

// Process List<Document> with parallel stream:

// 1. Filter valid documents (isValid())

// 2. Extract text (extractText())

// 3. Collect results maintaining order

**5. File Handling Prompts**

**Basic File Operations**

// Write code to:

// 1. Read all lines from file "input.txt"

// 2. Filter lines containing "ERROR"

// 3. Write filtered lines to "errors.txt"

**NIO File Operations**

// Using NIO.2 (Files, Paths), implement:

// 1. Recursive directory walk

// 2. Find all . files modified in last 7 days

// 3. them to "backup" directory

**File Serialization**

// Serialize List<Employee> to "employees.dat":

// 1. Use try-with-resources

// 2. Handle IOException

// 3. Implement custom serialVersionUID

**6. Combined Examples**

**Annotation + Streams**

// Process all methods in MyService class:

// 1. Find methods with @Cacheable annotation

// 2. Collect their names into Set<String>

// 3. Print summary statistics

**Collections + File Handling**

// Read CSV file "data.csv" and:

// 1. Parse into List<Person> (name,age,email)

// 2. Filter valid emails (regex validation)

// 3. Group by age brackets (0-18, 19-30, etc.)

**Interface + Streams**

// Implement a Transformer interface with:

// - transform(String input) method

// Then create List<Transformer> with 3 implementations:

// 1. Uppercase

// 2. Reverse

// 3. Duplicate

// Apply all transformations to "hello" using streams

**Best Practices for Effective Prompts**

1. **Be specific about requirements**:

// Bad: "Sort a list"

// Good: "Sort List<Employee> by hireDate descending using Comparator"

1. **Specify error handling needs**:

// "Read file with try-with-resources and handle FileNotFoundException"

1. **Request modern approaches**:

// "Use 8 streams instead of for-loops"

1. **Ask for optimization**:

// "Make this file operation memory-efficient for large files"

1. **Include edge cases**:

// "Handle null elements in the collection"

These prompts will help you generate **clean, efficient code** while leveraging modern language features with GitHub Copilot.

github copilot prompts for exception handling

**GitHub Copilot Prompts for Exception Handling**

Here's a comprehensive set of prompts to generate robust exception handling code in using GitHub Copilot:

**1. Basic Exception Handling**

**Try-Catch Block**

// Write a try-catch block to:

// 1. Read integer input from user

// 2. Handle InputMismatchException

// 3. Provide user-friendly error message

**Multiple Exception Handling**

// Handle both FileNotFoundException and IOException when:

// 1. Opening a file

// 2. Reading first line

// 3. Different error messages for each case

**2. Custom Exceptions**

**Creating Custom Exceptions**

// Create a custom InsufficientFundsException that:

// 1. Extends RuntimeException

// 2. Includes current balance and required amount

// 3. Has a constructor with descriptive message

**Using Custom Exceptions**

// In a BankAccount class, throw InsufficientFundsException when:

// 1. Withdraw amount exceeds balance

// 2. Include current balance in exception

// 3. Handle it in calling code

**3. Exception Propagation**

**Checked Exception Handling**

// Write a method that reads file content and:

// 1. Declares throws IOException

// 2. Caller should handle the exception

// 3. Include cleanup in finally block

**Unchecked Exception Strategy**

// Convert SQLException to custom unchecked DataAccessException

// 1. Wrap original exception

// 2. Add context information

// 3. Throw to caller

**4. Resource Management**

**Try-With-Resources**

// Use try-with-resources to:

// 1. Open FileInputStream and BufferedInputStream

// 2. Read file content

// 3. Automatically close resources

**Cleanup in Finally**

// Implement proper resource cleanup:

// 1. Open database connection

// 2. Execute query

// 3. Close connection in finally block

// 4. Handle null cases

**5. Advanced Exception Patterns**

**Exception Translation**

// Implement exception translation layer that:

// 1. Catches low-level IOExceptions

// 2. Converts to domain-specific exceptions

// 3. Preserves original cause

**Retry Mechanism**

// Implement retry logic for flaky operation:

// 1. Max 3 attempts

// 2. Exponential backoff

// 3. Stop on success or final failure

**6. Specific Exception Scenarios**

**Null Checking**

// Write null-safe method that:

// 1. Accepts List<String> parameter

// 2. Throws IllegalArgumentException if null

// 3. Returns concatenated strings

**Validation Logic**

// Validate user input with:

// 1. Check for null

// 2. Check string length (5-20 chars)

// 3. Throw custom ValidationException with all errors

**Thread Interruption**

// Handle thread interruption properly:

// 1. Check Thread.interrupted()

// 2. Clean up resources

// 3. Restore interruption status

**7. Exception Logging**

**Proper Exception Logging**

// Log exception with:

// 1. Full stack trace

// 2. Context information

// 3. Using SLF4J/Log4j

// 4. Different log levels for different exceptions

**User-Friendly Messages**

// Convert technical exceptions to:

// 1. User-friendly messages

// 2. Without exposing sensitive info

// 3. With support for localization

**8. Testing Exceptions**

**JUnit Exception Testing**

// Write JUnit test to verify:

// 1. Method throws IllegalArgumentException

// 2. With specific error message

// 3. When null input provided

**Mocking Exceptions**

// Using Mockito, test how code handles:

// 1. Simulated IOException

// 2. From mocked file operation

// 3. Verify recovery behavior

**Best Practices for Exception Handling Prompts**

1. **Be specific about exception types**:

// Bad: "Handle errors"

// Good: "Handle FileNotFoundException and provide fallback content"

1. **Request proper resource cleanup**:

// "Ensure database connection is closed even if exception occurs"

1. **Ask for context preservation**:

// "Wrap the low-level exception while adding business context"

1. **Specify recovery behavior**:

// "Retry the operation 3 times before failing"

1. **Include logging requirements**:

// "Log the exception at ERROR level with transaction ID"

**GitHub Copilot Prompts for JPA & Hibernate**

**1. Entity Modeling**

**Basic Entity Definition**

// Create a JPA entity for Product with:

// - id (auto-generated Long)

// - name (String, not null)

// - price (BigDecimal)

// - createdAt (LocalDateTime)

// - Use proper JPA annotations

**Entity Relationships**

// Model a bidirectional OneToMany between Department and Employee:

// - Department has many Employees

// - Employee belongs to one Department

// - Include cascade rules and fetch types

**Inheritance Mapping**

// Implement SINGLE\_TABLE inheritance for Payment with:

// - CreditCardPayment (cardNumber)

// - BankTransferPayment (accountNumber)

// - Common fields: amount, paymentDate

// - Use discriminator column

**2. Repository & DAO Patterns**

**Spring Data JPA Repository**

// Create a Spring Data JPA repository for User entity with:

// - Custom query to find by email

// - Method to find active users (status = 'ACTIVE')

// - Pagination support for findAll

**Custom Repository Implementation**

// Implement a custom repository method to:

// 1. Bulk update user status

// 2. Use EntityManager for batch processing

// 3. Add @Transactional and flush clear

**JPA Criteria API**

// Build dynamic query with Criteria API to:

// 1. Filter products by price range

// 2. Sort by name/price

// 3. Support optional category filter

**3. Query Optimization**

**JPQL Queries**

// Write JPQL query to:

// 1. Find orders with total > $100

// 2. Join with customer

// 3. Use positional parameters

// 4. Add pagination

**Named Entity Graphs**

// Define named entity graph for Order with:

// 1. Eager loading of orderItems

// 2. Lazy loading of customer

// 3. Use in repository method

**Batch Processing**

// Optimize bulk insert of 10,000 records:

// 1. Use Hibernate batch size

// 2. Configure JDBC batch

// 3. Manage transactions properly

**4. Transaction Management**

**Declarative Transactions**

// Implement service method with:

// 1. @Transactional with propagation

// 2. Read-only where appropriate

// 3. Custom rollback rules

**Optimistic Locking**

// Add optimistic locking to Product entity:

// 1. Version field

// 2. Handle OptimisticLockException

// 3. Retry mechanism

**Pessimistic Locking**

// Implement pessimistic locking for:

// 1. Inventory stock adjustment

// 2. Use PESSIMISTIC\_WRITE

// 3. With lock timeout

**5. Advanced Mapping**

**Embeddable Types**

// Create embeddable Address class with:

// - street, city, zipCode

// - Use in User entity

// - Add validation constraints

**JSON Column Mapping**

// Map JSON column to object:

// 1. Product with metadata (Map<String, Object>)

// 2. Use Hibernate @Type or JPA converter

**Temporal Data**

// Model entity with:

// 1. Creation timestamp (auto-set)

// 2. Update timestamp (auto-update)

// 3. Use 8 time API

**6. Performance Tuning**

**N+1 Solutions**

// Fix N+1 problem for:

// 1. Department with Employees

// 2. Use JOIN FETCH in query

// 3. Alternative with entity graph

**Second-Level Cache**

// Configure caching for:

// 1. Product entity (read-heavy)

// 2. Use Ehcache

// 3. Cache region settings

**Dirty Checking Optimization**

// Optimize update scenario:

// 1. Use @DynamicUpdate

// 2. Add @SelectBeforeUpdate

// 3. Measure performance impact

**7. Testing**

**Spring Data JPA Test**

// Write test for repository with:

// 1. @DataJpaTest

// 2. Test entity persistence

// 3. Verify query results

**Integration Test**

// Write transactional integration test:

// 1. Test service layer

// 2. Verify rollback behavior

// 3. Assert database state

**8. Audit Logging**

**Entity Auditing**

// Implement auditing for all entities:

// 1. CreatedBy, CreatedDate

// 2. LastModifiedBy, LastModifiedDate

// 3. Use Spring Data JPA auditing

**Hibernate Envers**

// Configure entity versioning with:

// 1. @Audited on Product

// 2. Custom revision entity

// 3. Query history records

**Best Practices for JPA Prompts**

1. **Be specific about annotations**:

// Bad: "Map relationship"

// Good: "Map OneToMany with lazy loading and orphan removal"

1. **Specify performance needs**:

// "Optimize query to avoid Cartesian product"

1. **Request transaction boundaries**:

// "Mark service method as transactional with read-only"

1. **Include edge cases**:

// "Handle detached entity merge scenario"

1. **Ask for modern approaches**:

// "Use JPA 2.2 stream support"