1. Product Requirements Document (PRD)

A **PRD (Product Requirements Document)** outlines **what needs to be built, why it matters, and how it helps users and the business**.

Key Components of a PRD

- **Why:** Justifies the need for the product (problem, feature request, opportunity).
- **What:** Describes the solution in a clear, structured way.
- **Good Requirements Should Be:**
    - Unambiguous, complete, verifiable, consistent, modifiable, traceable.

Example PRD Breakdown (DevFlow)

- **Impact:** Reduce sprint planning complexity and tool switching.
- **Target Users:** Software developers, project managers, engineering leaders.
- **Success Metrics:** Adoption rates, time reduction, accuracy improvements.
- **Functional Requirements:** Task management, sprint planning, Git integration, reporting.
- **Technical Requirements:** Performance benchmarks, scalability needs, security standards.
- **User Flow Examples:** Task creation, sprint planning.
- **Dependencies:** Frontend (React, TypeScript), Backend (Node.js, PostgreSQL), Infrastructure (AWS, Redis, CI/CD).
- **Milestones:** Feature releases and launch phases.
- **Risk Mitigation:** Technical, business, and security risks.

---

2. Design Document

A **Design Doc** provides a **technical blueprint** for implementing a system, ensuring alignment among engineers and future maintainability.

When to Write a Design Doc?

- **Major new features or systems** (e.g., adding real-time collaboration).
- **Significant architectural changes** (e.g., transitioning from monolith to microservices).
- **Complex technical decisions** (e.g., selecting a new database).

Key Components of a Design Doc

- **Goals & Non-Goals:** Clearly define scope.
- **System Architecture:** High-level diagrams and component breakdowns.
- **Detailed Design:** Data models, API designs, technical decisions, trade-offs.
- **Security Considerations:** Authentication, authorization, data encryption.
- **Monitoring & Alerting:** System health, business metrics, alert thresholds.
- **Migration & Rollout Plan:** Phased deployment strategy with rollback plans.

Example Trade-Off Decision

Database Selection

Option	Pros	Cons	Decision
PostgreSQL	Strong ACID compliance, JSON support, analytics capabilities	Higher memory use, complexity	✅ Selected (best for data integrity & querying)
MongoDB	Flexible schema, good for rapid iterations	Weak ACID compliance, risk of inconsistency	❌ Rejected
MySQL	Easy to manage, low memory usage	Limited JSON support, weaker concurrency	❌ Rejected

---

3. API Documentation

Best Practices for API Docs

- **Know Your Audience:** Keep it **developer-friendly**.
- **Scope Definition:** Cover **essential functionalities and edge cases**.
- **Consistent Formatting:** Use **structured headers, syntax highlighting**.
- **Examples & Common Errors:** Provide clear **input/output examples**.

Example API Documentation

```
# POST /api/v1/tasks
Request:
{
  "title": "Implement real-time updates",
  "description": "Add WebSocket support for task updates",
  "priority": 1
}
Response:
{
  "id": "uuid",
  "title": "Implement real-time updates",
  "status": "TODO",
  "created_at": "2025-01-12T..."
}
```

Errors & Handling: Document exceptions (`TypeError`, `ValueError`, `Unauthorized`).

Versioning & Changelog: Maintain updates (`v1.0.1 - Added new filter`).

---

4. Code Comments & Documentation

When to Write Comments?

✅ **DO Comment:**

- **Complex logic or algorithms**
- **Business rules & requirements**
- **Non-obvious fixes or workarounds**

```
# Using Floyd-Warshall algorithm for shortest paths
(O(V^3) complexity)
def find_shortest_paths(graph):
    """Computes shortest paths between all pairs of
vertices."""
```

❌ **DON'T Comment:**

- **Obvious operations** (e.g., `x = x + 1 # Increment x`)
- **Self-documenting code** (use meaningful function names instead).

Best Practices for Function Documentation

```
def calculate_order_total(items, discount_code=None):
    """
    Calculates the total price of an order.

    Args:
        items (List[Item]): List of order items.
        discount_code (str, optional): Promo code.

    Returns:
        float: Final order total after discounts.

    Raises:
        InvalidDiscountError: If discount code is
invalid.
        OutOfStockError: If items are unavailable.
    """
```

TODO & FIXME Comments

```
# TODO: Implement retry logic for API calls
# FIXME: Current implementation is not thread-safe
```

---

5. Engineering Best Practices

Common Pitfalls

❌ **Over-Engineering**

- **Overly detailed designs for simple problems**.
- **Premature optimization** (e.g., microservices for a small app).

❌ **Under-Engineering**

- **Skipping critical design decisions**.
- **Ignoring scalability & security from the start**.

✅ **Best Practices**

- **Keep it clear & concise**: Use **diagrams and examples**.
- **Plan for review**: Include **stakeholders and trade-offs**.
- **Balance flexibility and maintainability**.

Final Summary

1️⃣ **PRD ensures product clarity** with clear requirements, success metrics, and risk mitigation.
2️⃣ **Design Docs provide technical blueprints** for system architecture, API design, and security.
3️⃣ **API Docs improve developer experience** with structured documentation and real-world examples.
4️⃣ **Good commenting practices make code maintainable** with clear explanations and structured function docs.
5️⃣ **Avoid over- and under-engineering**: Focus on **scalability, performance, and simplicity**

1. Product Requirements Document (PRD) – Debugging & Improvement

Question:

You are reviewing the following **Product Requirements Document (PRD) for a new task management system**:

📌 **PRD Snippet:**

- **Target Audience:** "For everyone who needs to manage tasks."
- **Key Features:**
  - Users can create tasks.
  - Tasks can have deadlines.
  - Users receive notifications.
- **Success Metrics:** Improve productivity.
- **Technical Requirements:** Should work fast and store data.

✅ **Flaws in the Original PRD:**

1. **Vague Target Audience** – "For everyone" is **too broad**.
2. **Lack of Detail in Success Metrics** – "Improve productivity" is **not measurable**.
3. **Weak Technical Requirements** – "Work fast and store data" is **too ambiguous**.

✅ **Improved PRD Snippet:**

- **Target Audience:**
  - Software teams managing sprint workflows.
  - Freelancers tracking projects.
- **Success Metrics:**
  - 30% reduction in time spent managing tasks per week.
  - 20% increase in task completion rates.
- **Technical Requirements:**
  - Must handle **10,000 concurrent users**.
  - Support **real-time notifications with WebSockets**.

✅ **Why These Fixes Matter:**

- **Specificity ensures clarity** for developers and stakeholders.
- **Quantifiable success metrics** allow tracking progress.
- **Technical feasibility** is better defined.

✅ **Justification:**

- If the goal is **low-latency, real-time collaboration**, **OT is better** (since it's used in major editors).
- If **peer-to-peer collaboration is needed**, **CRDTs are a better choice**.

✅ **Alternative Approach:**

- **Hybrid Model**: **Use OT for real-time edits and CRDTs for offline syncing** (best of both worlds).

---

3. API Documentation – Debugging & Fixing Errors

Question:

You are reviewing the following **API documentation for a weather app**:

📌 **API Snippet:**

```
GET /weather
```

```
{
  "city": "Los Angeles",
  "temperature": 72,
  "conditions": "Sunny"
}
```

Solution:

✅ **Issues in the Original API:**

1. **Improper HTTP Method** – `GET /weather` should **not contain a request body**.
2. **Incomplete Response Format** – Missing **units, timestamps, and location details**.
3. **Lack of Versioning** – No `v1` or `v2`, which makes future updates difficult.

✅ **Improved API Documentation:**

```
GET /api/v1/weather?city=LosAngeles&units=metric
```

📌 **Response:**

```
{
  "city": "Los Angeles",
  "temperature": {
    "value": 22,
    "unit": "Celsius"
  },
  "conditions": "Sunny",
  "timestamp": "2025-03-14T15:30:00Z"
}
```

✅ **Why This is Better:**

- Uses **query parameters** instead of an invalid request body.
- Includes **units and timestamps** for clarity.
- Adds **versioning (`v1`)** for API maintainability.

Question:

The following Python function is used in an **e-commerce system to apply discounts**:

📌 **Original Code:**

```
def calc(t, d):
    return t - (t * d / 100)
```

🔍 **Tasks:**

1. Rewrite the function **with better naming and comments**.
2. Explain why **good documentation matters** in engineering teams.

Solution:

✅ **Improved Version:**

```
def apply_discount(total_price: float, discount_percent: float)
-> float:
    """
    Applies a percentage-based discount to the total price.

    Args:
        total_price (float): The original price before
discount.
        discount_percent (float): Discount percentage (0-100).

    Returns:
        float: The final price after applying the discount.

    Example:
        >>> apply_discount(100, 10)
        90.0
    """
    return total_price - (total_price * discount_percent / 100)
```

✅ **Why Good Documentation Matters:**

- **Better readability** – `apply_discount` is clearer than `calc(t, d)`.
- **Easier debugging** – Future developers **understand intent quickly**.
- **Prevents misuse** – Function explains **expected input/output behavior**.

Question:

A junior developer suggests **using microservices** for a **simple to-do list app**.

📌 **Their Justification:**

- Microservices are **scalable**.
- Each feature (tasks, users, notifications) should be **its own service**.
- Helps **prevent tech debt** early.

Solution:

✅ **Flaws in Using Microservices for a To-Do List App:**

- **Overhead** – Managing multiple services **adds complexity** (deployment, monitoring).
- **Unnecessary for Small Apps** – **Monoliths** are simpler and easier to maintain.
- **Premature Optimization** – Scaling should be done **when needed**, not preemptively.

✅ **Simpler Alternative:**

- Use a **single backend API** with a **modular architecture**.
- If scaling is needed later, **split into services gradually**.

✅ **When Microservices Make Sense:**

- When the app needs **independent scaling** (e.g., Netflix, Amazon).
- If **different teams** manage different services.
- When services have **different storage or compute needs**.
