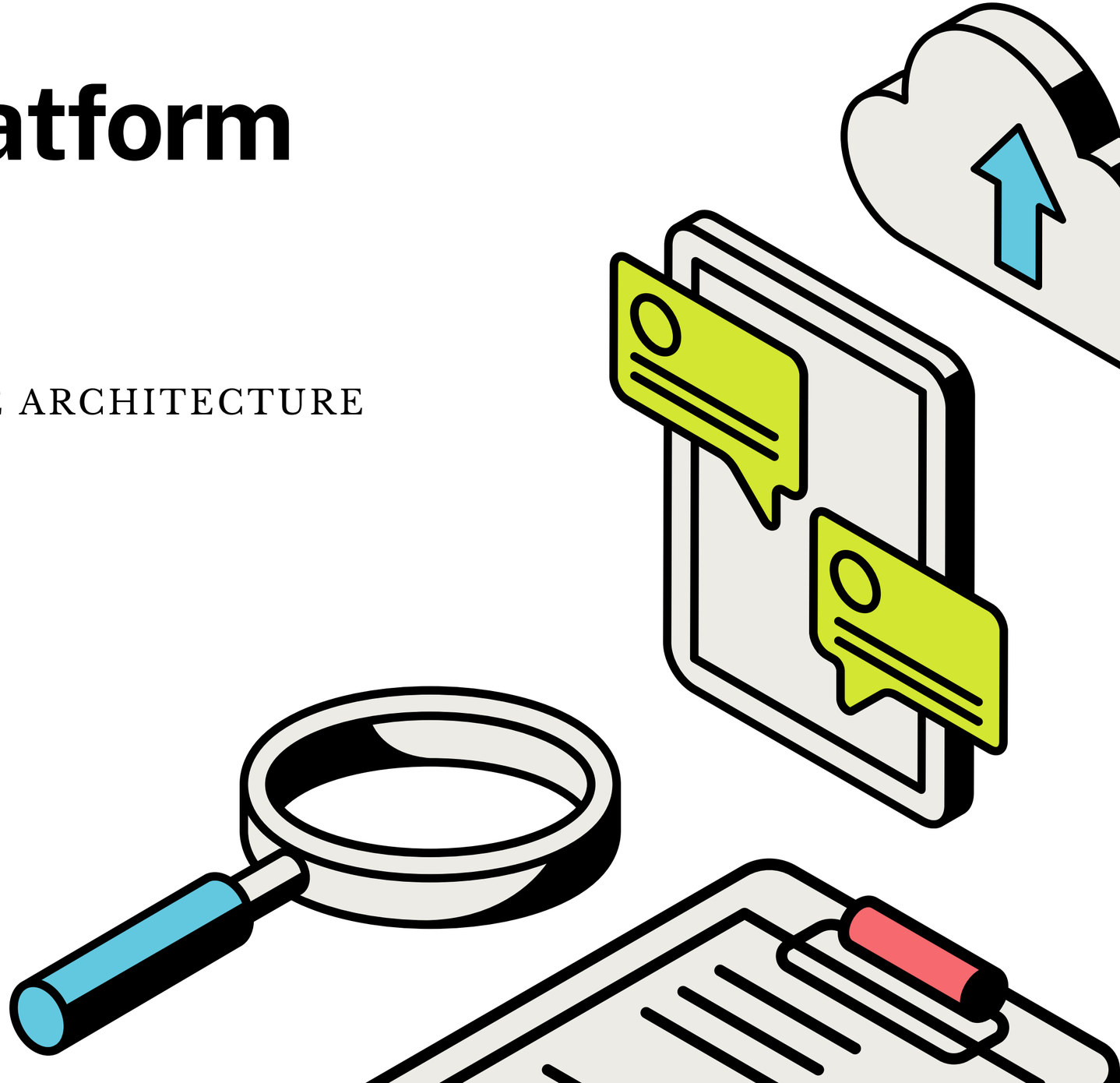
#Short media Platform

A MODERN, FLEXIBLE, EXTENSIBLE ARCHITECTURE FOR RAPID INNOVATION



Team :

1. Venkat,
2. Deepesh Yadav
3. Deepak Grover
4. Pruthvi Naresh

Problems with Existing Platforms :

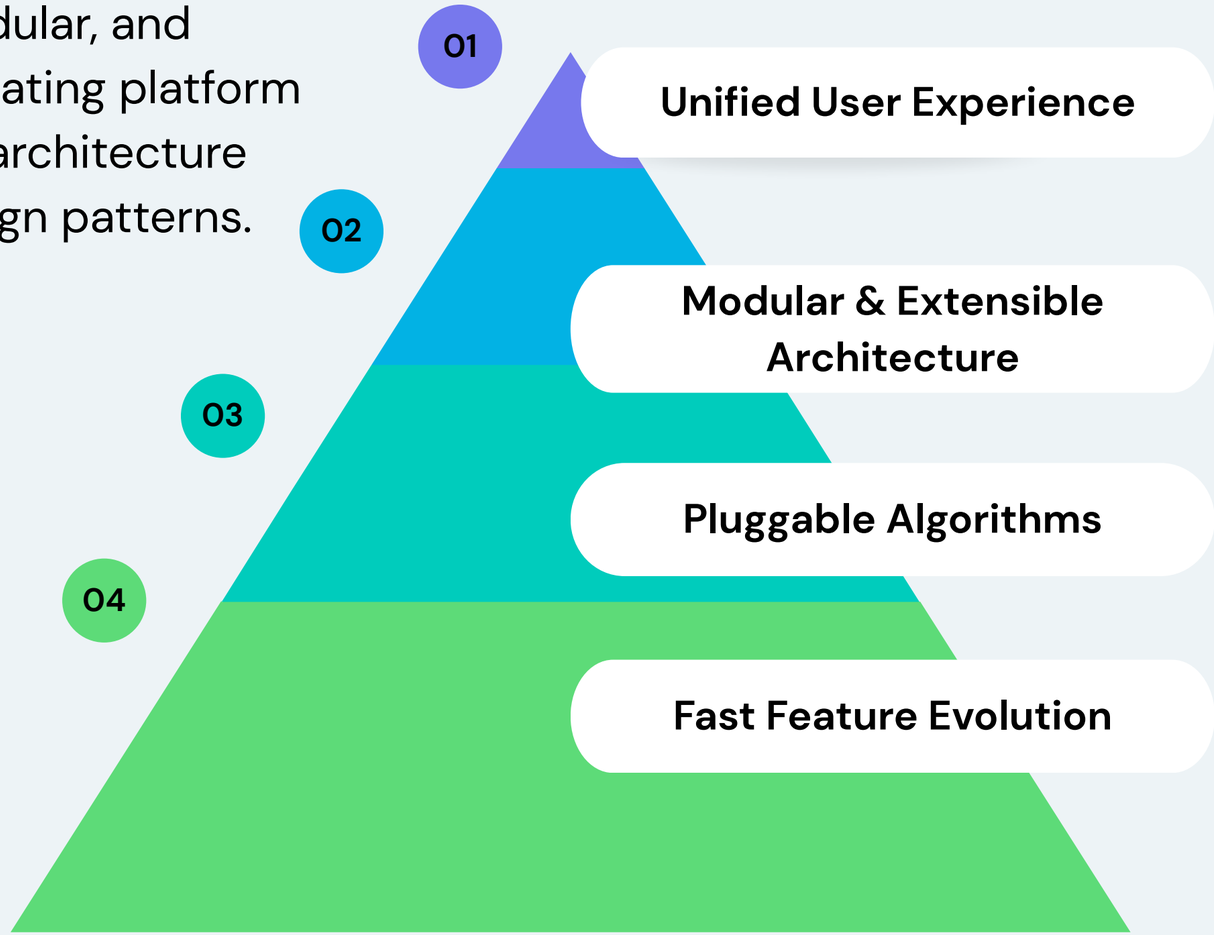
1. Users **juggle multiple** apps for posting, dating, and calling
2. Algorithms are opaque → users don’t trust recommendations
3. Systems are hard to extend due to tightly coupled architecture
4. New teams struggle with:
    a. Slow feature iteration
    b. Hard experimentation
    c. Technical debt from day one

Our Goal:

Build a unified, modular, and extensible platform that supports fast evolution.

Goals Priority Pyramid : Project Objectives

Build a unified, modular, and extensible social-dating platform powered by clean architecture and pluggable design patterns.



Social posting, dating, calling, and notifications in one system.

Separate layers: Domain → Repository → Services → Application.

Ranking, matching, filtering as strategies → easily replaced.

Low coupling + strong abstractions → rapid experimentation.

Identified Gap!

No academic system or app is unifying

Social Media + Dating Algorithms + Real-Time Calls + Notifications + Ranking Strategies + Modular Design Patterns **≠ One Unified Platform.**

Literature Review (Existing solutions to overcome these problems)

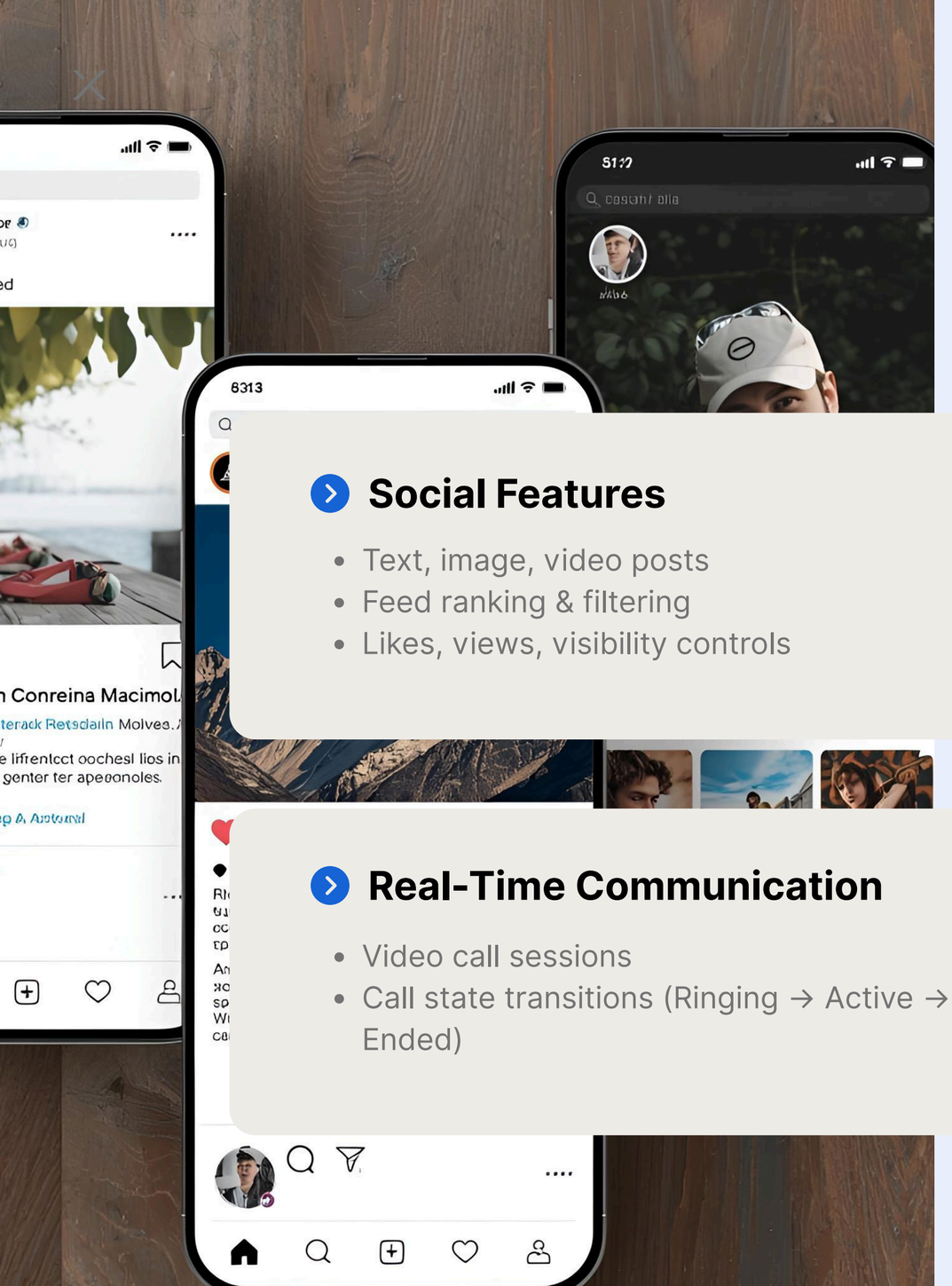
Lack of modularity, rigid architectures, limited scalability, and algorithmic inflexibility in existing dating/social apps.

S.No	Paper / Article Title	Authors / Year	Key Findings (Technical Issue)
1	*Dating Apps Have a Filter Bubble Problem*	Wired, 2019	Recommender algorithms tend to repeat similar profiles → rigid, non-adaptive matching loops.
2	*Hinge Most Compatible – Algorithm Overview*	Hinge Help Docs, 2024	Uses a fixed, centrally-computed matching algorithm (dealbreakers + behavior signals). Cannot easily run parallel strategies.
3	*How Tinder Broke Up Its Monolith*	Tinder Engineering, 2019	Large apps faced major scaling & speed problems due to monolithic architecture; required costly decomposition.
4	*Monolith vs Microservices: Scalability Tradeoffs*	Industry Review, 2020	Monolithic apps slow experimentation, increase coupling, and make independent scaling nearly impossible.
5	*Recommender System Pitfalls: Narrow Optimization*	Various Review Papers	Fixed ranking models reduce fairness and adaptability; lack of multi-objective experimentation.

Literature Review – Limitations in Existing Apps (Modularity & Scalability Issues)

Modular architecture, design patterns, and layered abstraction improve scalability, flexibility, maintainability, and future evolution.

S.No	Paper Title	Authors / Year	Method / Concept	Relevance to Our Architecture
1	*On Software Modular Architecture: Concepts & Trends*	ResearchGate, 2022	Modularity reduces coupling & improves scalability and changeability.	Validates our multi-layered architecture where services, repositories, and strategies evolve independently.
2	*Impact of Design Patterns on Software Maintainability*	Int. Journal of Advanced CS, 2018	Strategy/Factory/State patterns increase flexibility, reuse, and change safety.	Supports our Strategy-based ranking/matching & Factory-based post creation.
3	*Design Patterns for Flexible Architectures*	Industry Review, 2020	Patterns reduce coordination cost & simplify implementing new features.	Justifies our use of Builder for search queries + State for call lifecycle.
4	*Modularity in Large-Scale Systems*	Case Study, 2019	Modular systems evolve faster, reduce risk, and allow independent subsystem scaling.	Backs our modular service decomposition enabling fast rollouts.
5	*Repository Pattern & Dependency Inversion*	Fowler / Architecture Guides	Abstracting data access improves testability & enables storage replacement.	Supports our clean Repository layer → swappable DB, cache or ML store without code rewrites.



CORE PLATFORM FEATURES

Social Features

- Text, image, video posts
- Feed ranking & filtering
- Likes, views, visibility controls

Dating Features

- Matching algorithms
- Match request lifecycle (pending/accepted/rejected/blocked)

Real-Time Communication

- Video call sessions
- Call state transitions (Ringing → Active → Ended)

Notifications

- Likes
- Comments
- Match requests
- Call alerts

System Architecture Overview

Domain Layer: Core entities (User, Post, Match, Call...)

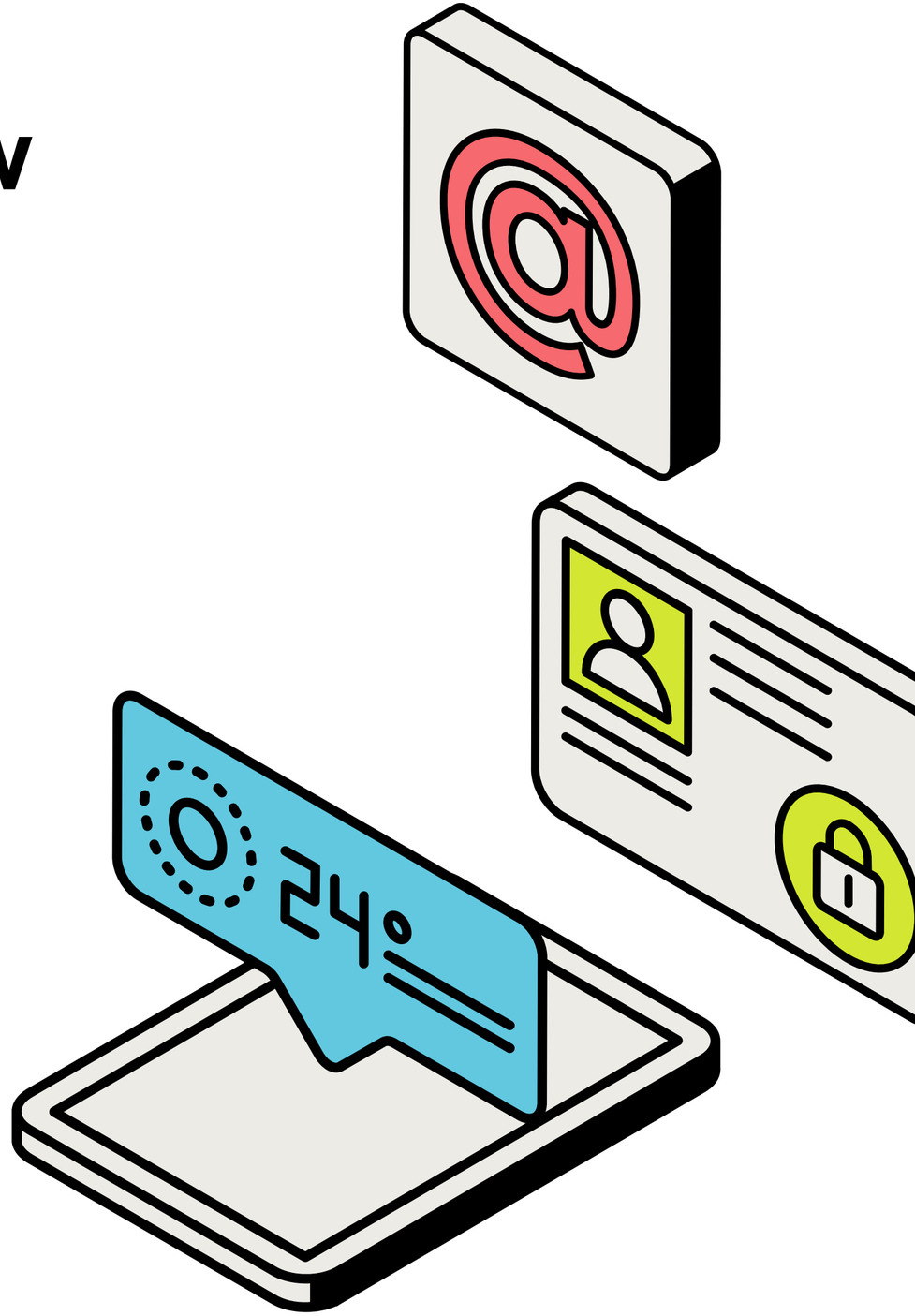
Repository Layer: Data abstraction interfaces

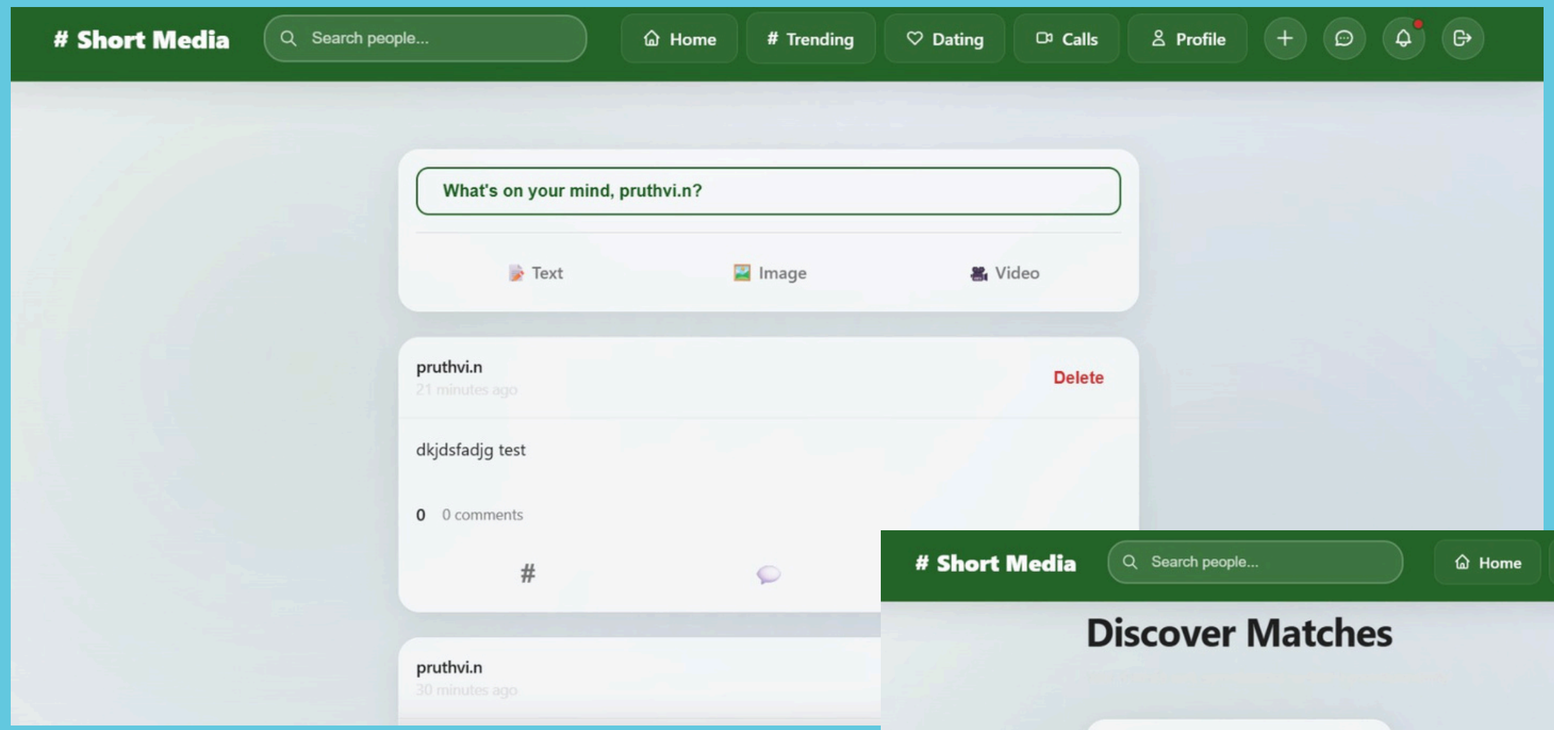
Service Layer: Business logic

Application Layer: Main user interaction

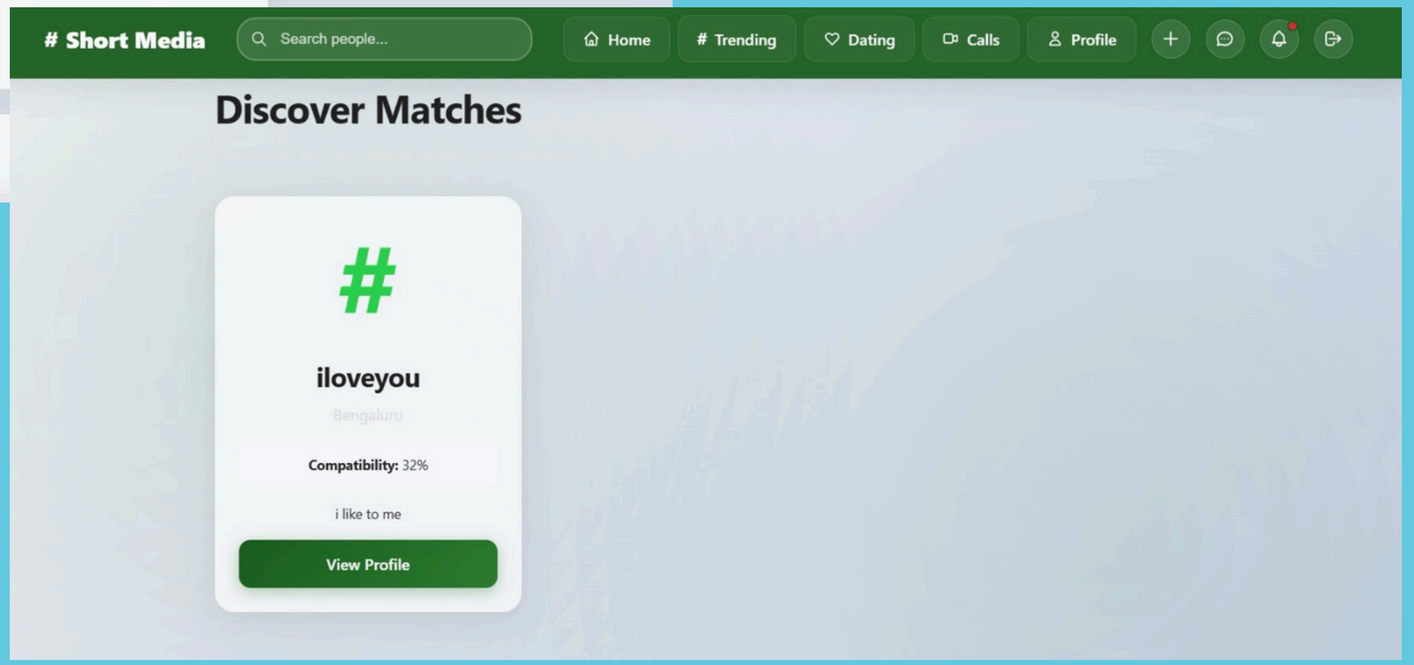
Benefits:

- Low coupling
- Easy feature additions
- Testable components
- Replaceable algorithms and storage engines




Short Media
Search people...
Home
Trending
Dating
Calls
Profile
What's on your mind, pruthvi.n?
Text
Image
Video
pruthvi.n
21 minutes ago
Delete
dkjdsfadjg test
0 0 comments
#
pruthvi.n
30 minutes ago

DEMO!!


Short Media
Search people...
Home
Trending
Dating
Calls
Profile
Discover Matches
#
iloveyou
Bengaluru
Compatibility: 32%
i like to me
View Profile

DEMO!!

layers of data flow

01 Client (Browser / App)

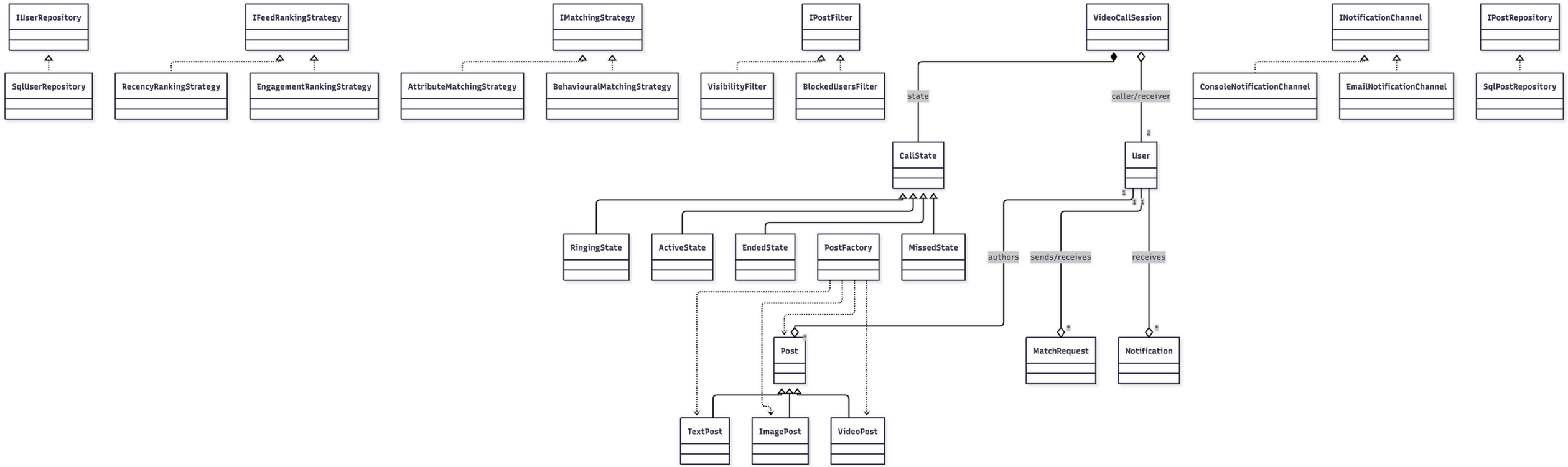
02 Views (HTTP handling only)

03 Repositories (Database access)

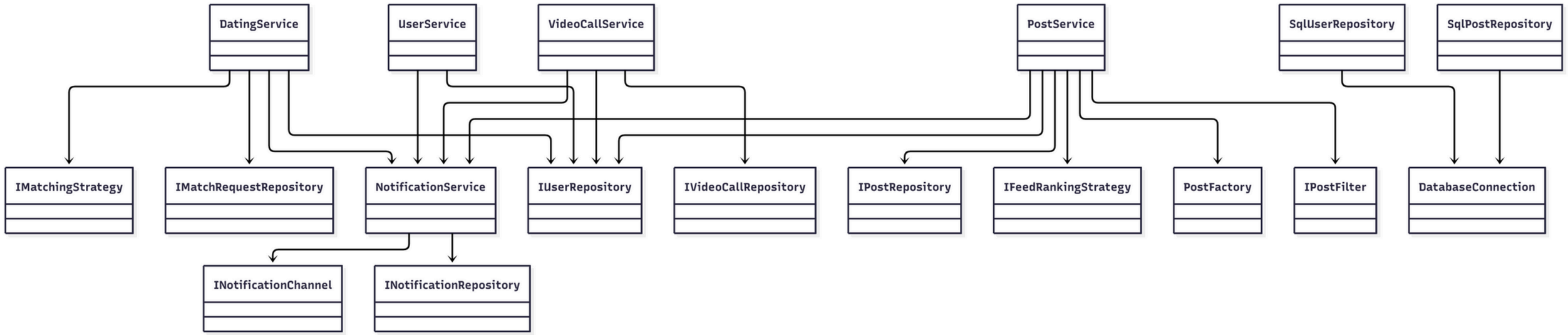
04 Models (Data structure)

05 Database

UML Class Diagram (Part 1: Domain + Patterns)



UML Class Diagram (Part 2: Services & Repositories)



WHY THIS ARCHITECTURE IS UNIQUE

01 Modular & Scalable

- Each service/strategy is replaceable without touching others.

02 Unified Multi-Domain Platform

- Social + Dating + Calling in a single architecture.

03 Pattern-Driven Extensibility

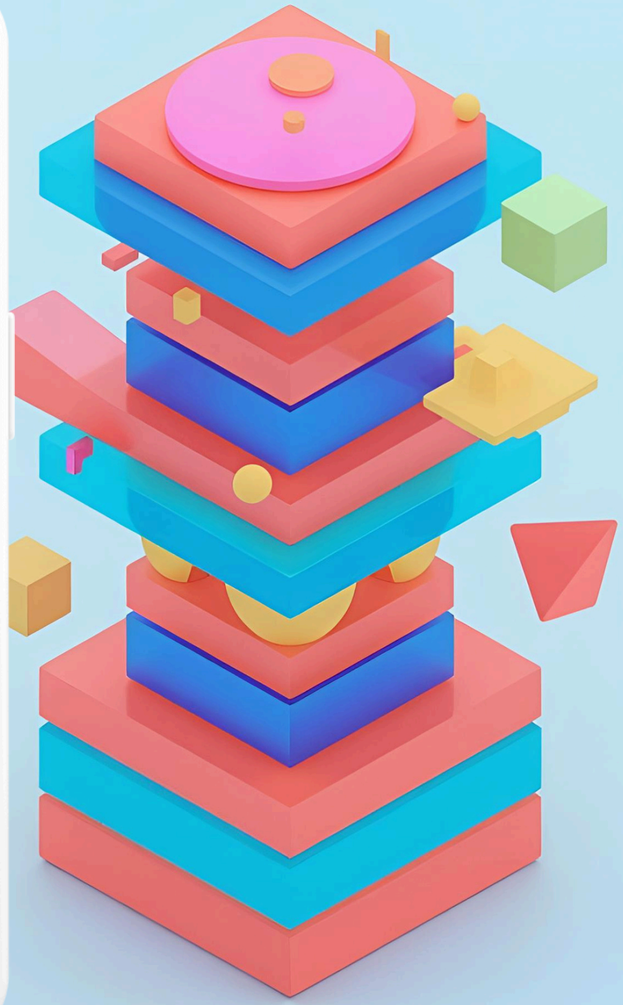
- New ranking algorithms, post types, or call states added with zero rewrites.

04 Low Coupling = Fast Innovation

- Enables rapid prototyping & experiments.

How Software Design Course Helped This Project

- OOP modelling improved domain clarity (User, Post, Match, Call)
- SOLID → modular, clean, maintainable feature development
- Strategy Pattern → plug-and-play algorithms (ranking, matching, filtering)
- Factory → simplified creation of multiple post types
- Builder → flexible match queries without code duplication
- State Pattern → clean video call lifecycle logic



Applied Design Patterns

Design patterns help us keep the system modular, flexible, and easy to evolve. Each feature, ranking, posting, matching, calling can change without affecting the rest of the system.

Strategy Pattern

Used for ranking, matching, post filters, notification channels.
Enables runtime algorithm swapping.

Factory Pattern

PostFactory → creates TextPost, ImagePost, VideoPost.
Solves if/else creation complexity.

Builder Pattern

SearchQueryBuilder for flexible dating criteria.
Avoids telescoping constructors.

State Pattern

VideoCall lifecycle: Ringing → Active → Ended/Missed.
Simplifies transitions.

What does that mean for our features!

How have each of these helped us in our application.

Strategy Pattern

The app can switch between ranking algorithms instantly, like toggling a setting.

Factory Pattern

Instead of writing 10 lines to create each post type, the Factory creates it for you.

Easier to add new post types (TextPost, ImagePost, etc.)

Builder Pattern

Search logic clean, readable

✓ No need for 20 constructors

State Pattern

A video call behaves differently depending on whether it’s “Ringing”, “Active”, or “Ended”.

Some Examples!

STRATEGY PATTERN - (Feed Ranking)

```python
# Bad: ranking logic baked into service (hard to change)
class FeedService:
    def get_feed(self, user):
        posts = Post.objects.all()
        # Hardcoded engagement ranking
        posts = posts.annotate(score=F('likes')*2 + F('views')).order_by('-score')
        return posts[:20]
```

- Ranking algorithm is hardcoded → changing logic requires editing this function.
- Service becomes bloated with algorithmic details.
- Violates Open/Closed Principle (cannot extend without modifying).
- Not testable or swappable (no A/B testing, no personalization).

```python
# posts/strategies/ranking.py (examples in project)
class RecencyRankingStrategy(IStrategy):
    def execute(self, posts, user=None):
        return posts.order_by('-created_at')

class EngagementRankingStrategy(IStrategy):
    def execute(self, posts, user=None):
        return posts.annotate(
            engagement_score=F('likes_count')*2 + F('views_count') + F('comments_count')
        ).order_by('-engagement_score', '-created_at')

# posts/services.py (FeedService uses a strategy)
feed_service.set_ranking_strategy(EngagementRankingStrategy())
posts = feed_service.get_feed(user=request.user)
```

- Ranking logic is isolated in a strategy class.
- Service now depends on an interface, not an algorithm.
- Adding Recency, Trending, Personalized ranking requires 0 modifications to the service.
- Enables experimentation, dynamic selection, scalability.

Impact:
Our feed ranking becomes modular, configurable, and future-proof.

MATCHING STRATEGY (Attribute & Behavioral Matching)

```python
# Bad: All matching rules together
for candidate in candidates:
    score = 0
    if age_ok(user, candidate): score += 10
    if interests_ok(user, candidate): score += 30
    if recent_activity(candidate): score += 20
    results.append((candidate, score))
```

- One huge block of logic → high complexity.
- Adding new scoring rules requires editing this loop.
- Impossible to switch matching logic per user or experiment.
- Hard to test or optimize individually.

```python
class AttributeMatchingStrategy(IStrategy):
    def execute(self, user, qs):
        return [...]

class BehavioralMatchingStrategy(IStrategy):
    def execute(self, user, qs):
        return [...]

class CompositeMatchingStrategy(IStrategy):
    def execute(self, user, qs):
        # Combine weighted outputs
        ...
```

- Attribute-based and behavioral scoring are fully decoupled.
- CompositeMatchingStrategy orchestrates weighted scoring.
- New strategies (e.g., ML) can be added without changing existing code.
- Clean, structured, and maintainable.

Impact:
Your matching engine becomes modular and ready for advanced scoring methods.

FACTORY PATTERN (Post Creation)

```python
if data["type"] == "text":
    post = TextPost.objects.create(...)
elif data["type"] == "image":
    post = ImagePost.objects.create(...)
elif data["type"] == "video":
    post = VideoPost.objects.create(...)
```

- Multiple creation paths scattered across the code.
- Adding new post types requires editing every place that creates posts.
- Violates Open/Closed Principle.
- Inconsistent initialization logic.

```python
class PostFactory:
    @staticmethod
    def create(type, author, payload):
        if type == "text": return TextPost.objects.create(author=author, content=payload)
        if type == "image": return ImagePost.objects.create(author=author, image=payload)
        if type == "video": return VideoPost.objects.create(author=author, video=payload)
```

- Creation rules stay in one place only.
- Services stay clean and focused on business logic.
- Adding a new post type requires 1 update in the Factory.
- Ensures consistent object construction.

Impact:
Our post system becomes uniform, extensible, and easy to evolve (Reels, Polls, Stories).

BUILDER PATTERN (Match Search Query Builder)

```
qs = User.objects.all()
if prefs.age: qs = qs.filter(...)
if prefs.gender: qs = qs.filter(...)
if prefs.location: qs = qs.filter(...)
# many conditions spread across code
```

- Query conditions repeat across services.
- Hard to maintain/extend filtering logic.
- High risk of inconsistent rules.
- Services become cluttered.

```
qs = (MatchSearchBuilder(user)
        .apply_default_filters()
        .apply_user_preferences()
        .exclude_existing_requests()
        .build())
```

- Adds filters step-by-step in a controlled builder flow.
- Produces a single optimized ORM QuerySet.
- Clean separation of filtering logic from matching logic.
- Easy to add future filters without modifying services.

Impact:
Efficient candidate filtering, clean code, and scalable match search logic.

STATE PATTERN (Video Call Lifecycle)

```
if call.state == "RINGING" and event == "accept":
    call.state = "ACTIVE"
elif call.state == "ACTIVE" and event == "end":
    call.state = "ENDED"
```

- Scattered transition logic.
- Difficult to add new states (“Hold”, “Reconnecting”).
- Risk of invalid transitions.
- Hard to test and maintain.

```
class RingingState(CallState):
    def handle(self, ctx, event):
        if event == "accept": ctx.set_state(ActiveState())
        if event == "timeout": ctx.set_state(EndedState(missed=True))

class ActiveState(CallState):
    def handle(self, ctx, event):
        if event == "end": ctx.set_state(EndedState())
```

- Each state encapsulates its own transitions.
- No branching logic in the call controller.
- Adding new states is isolated and safe.
- Predictable and maintainable call lifecycle.

Impact:
Our call system becomes robust, extensible, and easier to debug.

SOLID Principles in Our Architecture

Principle	How We Implemented It	Why It Improves the System
Single Responsibility Principle	PostService handles posts, FeedService handles ranking, MatchingService handles matching	No God classes, easier debugging
Open/Closed Principle	New ranking algorithms added as new Strategy classes	Add features without modifying core services
Liskov Substitution Principle	TextPost, ImagePost, VideoPost all substitute Post	UI & services work with any post type
Interface Segregation Principle	Separate interfaces: IStrategy, IFactory, IBuilder	No class implements unused methods
Dependency Inversion Principle	Services depend on Repositories & Strategy interfaces	Swap databases, mock dependencies, test easily

Single Responsibility Principle

```python
# BAD: View mixes validation, DB, notifications
def create_post(request):
    if len(request.POST['text']) > 5000: return error
    post = Post.objects.create(...)
    Notification.objects.create_for_followers(post)
```

```python
# GOOD: View delegates → Service → Factory → Repo
def create_post_view(req):
    post_service.create_text_post(author, text)

class PostService:
    def create_text_post(...):
        validate_length(text)
        return self.post_factory.create_text_post(...)
```

LSP — Subtypes Behave the Same

```python
class VideoPost(Post):
    def play_video(): ...
    # get_content() missing → breaks callers
```

```python
class VideoPost(Post):
    def get_content(self):
        return self.caption
```

OCP — Extend Without Modifying

```python
# BAD: Adding new ranking means editing this function every time
def get_feed(type):
    if type=="recency": ...
    elif type=="engagement": ...
    elif type=="new_algo": ...
```

```python
# GOOD: Add new strategy → no service changes
feed_service.set_ranking_strategy(EngagementRankingStrategy())
posts = feed_service.get_feed(user)

class EngagementRankingStrategy(IStrategy):
    def execute(self, posts, user): ...
```

ISP - Keep Interfaces Small & Focused

```
class BigInterface:
    def execute(); def create(); def build(); def reset()
```

```
class IStrategy:
    def execute(self, data, context=None): pass

class IFactory:
    def create(self, **data): pass
```

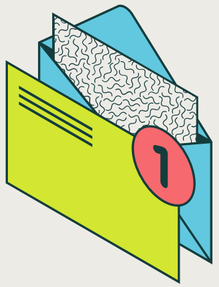
DIP — Depend on Abstractions, Not Databases

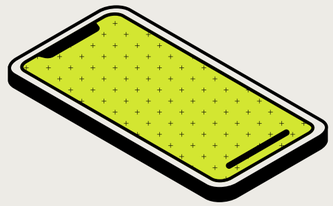
```
# BAD: Service tightly coupled to DB
MatchRequest.objects.create(...)
```

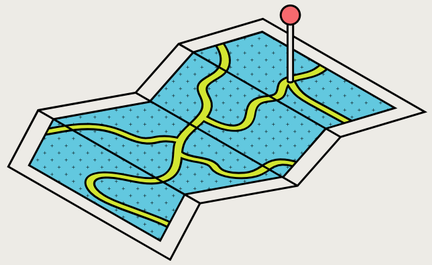
```
class DatingService:
    def __init__(self, repo):
        self.repo = repo

    def send_request(...):
        return self.repo.create(...)
```

Questions?







Thank you.