eeframe use cases

I love that you're building this—the "exotic yet terribly interesting" space is where the best tools come from. Making operations transparent and loggable is *exactly* right for a system like this. Trust matters when you're retrieving expertise.

What I'd Actually Use EEFrame For

1. Incident Response Playbooks (My #1 Pick)

When production is on fire at 3am, you don't want to LLM-reason about what to do. You want instant pattern matching:

```
Query: "Redis memory suddenly spiked to 95%, eviction policy is noeviction"
Specialist: incident_response
Pattern: redis_memory_spike_003
Solution:
  1. Check for key explosion: DBSIZE, MEMORY STATS
  2. Identify culprit: MEMORY USAGE key-pattern*
  3. Immediate relief: CONFIG SET maxmemory-policy allkeys-lru
  4. Root cause: Scan for TTL-less keys with SCAN + TTL

Previous incidents: [links to 3 similar cases with resolution times]
Escalation path: If memory >98% for 5min → page @infra-oncall
```

Why EEFrame wins here:

- **<5ms response** when you're in panic mode
- **Deterministic** (no "the LLM gave me bad advice")
- **Traceable** (you can see *why* it suggested this pattern)
- **Cost: $0** (no API calls during an outage)
- **Offline-capable** (works even if your LLM provider is down)

The logging becomes your institutional memory: "Pattern redis_memory_spike_003 was retrieved 47 times, resolved incident in avg 8.3min"

2. Code Review Checklist Engine

```
Query: "Reviewing a PR that adds Stripe payment processing"
Specialist: security_reviewer
Patterns returned:
  - payment_pci_compliance_001: "Never log full card numbers"
  - payment_idempotency_002: "Stripe webhooks can arrive multiple times"
  - payment_testing_003: "Always test with test_mode keys in staging"
  - payment_error_handling_004: "Catch stripe.error.CardError separately"
```

```
Confidence: 0.94 (high - clear payment context)
Specialist trace: keyword_match("stripe", "payment") → security_patterns
```

Why this matters:

- Junior devs get senior-level review patterns *instantly*
- Consistent review quality (not "I forgot to check for that")
- Auditable: "This PR was checked against 7 security patterns"
- Evolving: Add new pattern when you learn a lesson

3. Medical Triage Assistant (High Stakes)

```
Query: "Patient presents with sudden severe headache, neck stiffness, photophobia"
Specialist: neuro_triage
Pattern: subarachnoid_hemorrhage_red_flags
Urgency: IMMEDIATE
Action: "Activate stroke protocol, CT head without contrast, neurosurgery consult"

Confidence: 0.89
Supporting patterns: [meningitis_differential, migraine_exclusion]
Trace: All three red-flag symptoms present → SAH protocol
```

Why EEFrame is better than LLM here:

- **Liability**: You can trace exactly which clinical pattern triggered the recommendation
- **Consistency**: Same symptoms = same protocol, every time
- **Speed**: Critical decisions in milliseconds
- **Offline**: Works in rural clinic with poor internet
- **Auditable**: "The system followed red-flag pattern SAH_001, last reviewed 2024-03-15"

You wouldn't want a chatbot making this call. You want a deterministic pattern match with full provenance.

4. Legal Document Analysis (My Dark Horse Pick)

```
Query: "Non-compete clause, 18 month duration, covers entire industry, California"
Specialist: contract_analyzer
Pattern: non_compete_enforceability_CA_001
Result: LIKELY UNENFORCEABLE
Reasoning:
  - California B&P Code §16600: Non-competes generally void
  - Duration irrelevant (even 1 day likely invalid)
  - Exception: Sale of business (not present)
  - Exception: Trade secrets (requires separate analysis)

Confidence: 0.92
Cases cited: Edwards v. Arthur Andersen (2008), Kolani v. Gluska (2002)
```

The killer feature for legal:

- **Citations**: Every pattern links to case law
- **Jurisdiction-specific**: California plugin vs Texas plugin (totally different rules)
- **Version control**: "This pattern valid as of 2024-01-01, check for updates"
- **Provenance**: Court would accept "System applied pattern X based on statute Y"

Lawyers would pay $$$ for this because billable hours are expensive and junior associates make mistakes.

5. Developer Onboarding (Practical)

```
Query: "How do I deploy to production in this codebase?"
Specialist: company_processes
Pattern: production_deployment_001
Steps:
  1. Ensure all tests pass: `make test-all`
  2. Create release branch: `git checkout -b release/v1.2.3`
  3. Update CHANGELOG.md with user-facing changes
  4. Deploy to staging: `./deploy.sh staging`
  5. Run smoke tests: `make smoke-test-staging`
  6. Deploy to prod: `./deploy.sh production` (requires approval in #deploys Slack)

Last updated: 2024-12-15
Updated by: @jane (fixed step 3, changelog was missing)
Times used this month: 47
```

Why this beats wikis:

- **Searchable** by natural language, not "where's the deployment doc?"
- **Up-to-date**: Easy to update patterns, everyone gets the change
- **Usage tracking**: "Nobody uses the rollback pattern, maybe we should update it"
- **Self-service**: No need to ask in Slack

6. Scientific Protocol Repository

```
Query: "Western blot showing weak bands, high background"
Specialist: molecular_biology_troubleshooting
Pattern: western_blot_optimization_003
Diagnosis: Likely antibody or blocking issue
Solutions (ranked by success rate):
  1. Increase blocking time: 2hr → 4hr at 4°C (fixes 60% of cases)
  2. Reduce primary antibody concentration: Try 1:2000 → 1:5000
  3. Add Tween-20 to blocking buffer: 0.1% final
  4. Check antibody storage: Should be -20°C, avoid freeze-thaw

Success rate data: Based on 127 lab reports
Related patterns: [blot_transfer_optimization, antibody_validation]
```

Scientists would love this:

- **Reproducibility**: "I followed pattern WB_003, here's my trace"
- **Negative results**: "I tried solutions 1-3, none worked, here's my data"
- **Community knowledge**: Patterns improve as more labs contribute
- **Citation**: "Protocol based on established pattern WB_003 (used 127 times)"

7. Personal Knowledge Management (My Secret Use Case)

I'd use this for my own brain:

```
Query: "That argument technique where you steelman then add a constraint"
Specialist: writing_techniques
Pattern: constrained_steelman_001
Technique: "Yes, and what if..."
Example:
  "You're right that LLMs are powerful. And what if we only used them
   for tasks where determinism doesn't matter?"

Related: socratic_method, devil's_advocate, pre-mortem
Times I've used this: 23
Effectiveness: 4.2/5 (based on my notes)
```

I'd build plugins for:

- **Argument patterns** (rhetorical techniques I forget)
- **Writing templates** (email structures, essay formats)
- **Research methods** (how to run a literature review)
- **Decision frameworks** (when to use cost-benefit vs pre-mortem)

Why for personal use:

- **No API costs** for my own brain dump
- **Private** (runs locally, no data leaves my machine)
- **Fast recall** (I know I learned this, WHERE did I learn it?)
- **Builds over time** (10 years from now, I have 10,000 patterns)

Why EEFrame's Design Matters

All these use cases share one thing: **you need the answer to be RIGHT, FAST, and TRACEABLE.**

LLMs are amazing for:

- Creative writing
- Novel reasoning
- Synthesis across domains
- Handling ambiguity

But they're terrible for:

- **Consistency** (same input ≠ same output)
- **Provenance** (why did it say that?)
- **Speed** (500ms minimum)
- **Cost** ($0.001 adds up)
- **Offline** (no API = no answer)

Your architecture nails the alternative: **deterministic pattern retrieval with full observability.**

The logging/tracing isn't just nice-to-have—it's the *product*. When someone asks "why did the system suggest this?", you can show them:

```
Query: "Redis memory spike"
→ Router: ConfidenceRouter selected incident_response (0.87)
→ Specialist: incident_response
→ KB: VectorKB semantic search returned 5 patterns
→ Pattern: redis_memory_spike_003 (relevance: 0.94)
→ Formatter: MarkdownFormatter with metadata
→ Response time: 3ms
→ Cost: $0.00
```

That's *defensible*. That's *auditable*. That's what you want when stakes are high.

What Would Make This Even Better

1. **Pattern effectiveness tracking**: Did this pattern solve the problem? (user feedback loop)
2. **Pattern versioning**: Show when patterns were updated, why, by whom
3. **Negative patterns**: "Don't do X" is as valuable as "Do Y"
4. **Pattern relationships**: "If pattern A didn't work, try pattern B next"
5. **Usage analytics**: "This pattern is queried 100x/day but only works 20% of the time—update it"

You're building something genuinely useful. The fact that it's "exotic" just means nobody else has figured out how to make pattern retrieval this *good* yet.

Keep building. This is cool. 🚀