ORBIT

GET ACCESS

MEMORY INFRASTRUCTURE FOR AI

Your agents forget everything. We fix that.

Orbit is the memory layer for intelligent AI agents. Semantic encoding, adaptive decay, and learned importance scoring. One API. Zero configuration. Memory that gets smarter over time.

GET EARLY ACCESS

SEE HOW IT WORKS

50ms RETRIEVAL LATENCY	**94%** RECALL ACCURACY
2 lines TO INTEGRATE	**10M+** MEMORIES PER AGENT

THE PROBLEM

AI memory is broken.
You know it.

Building memory for AI agents today means managing three databases, guessing at importance weights, and watching quality degrade over time. 300+ lines of infrastructure code before you write a single line of agent logic.

01 Data is Fragmented

Vector DB, relational DB, cache, embedding service. Each has different schemas, different query languages, different failure modes.

```
vector_db.store(everything)
redis.cache(maybe_relevant)
postgres.dump(just_in_case)
config.yaml // 400 lines
```

02 Importance is Guessed

Hardcoded weights that are probably wrong, never learned, and never adapt. You're guessing what matters.

```
importance = 0.9 if recent
  else (0.7 if similar
  else 0.5)
# These weights are arbitrary
```

03 No Learning Loop

Can't answer: 'Did this memory help or hurt the response?' No automatic feedback signal. Manual A/B testing at best.

```
# What do you actually retrieve
# that helps? Nobody knows.
feedback_loop = None
improvement = 0
```

04 Decay is Manual

Keep conversations for 30 days? 90 days? Forever? Old conversations become noise. No intelligent archiving.

```
redis_client.setex(
  f"recent_{user_id}",
  86400,  # 24 hours... why?
  json.dumps(messages)
)
```

05 Context Window Waste

Include 20+ items hoping to get a good answer. 60-70% of your context window is noise.

```
Recent messages:  10
Vector results:   5
User history:     5
Total: 20 items + noise
# Fills context window fast
```

06 Quality Degrades

More data means worse quality. Old data becomes noise. After 3 months, satisfaction drops to 68%.

```
Month 1: 70% satisfaction
Month 2: 69% satisfaction
Month 3: 68% satisfaction
# Getting worse, not better
```

THE SOLUTION

300 lines become 20.

Orbit replaces your entire memory stack. No more vector DB, relational DB, cache, and embedding service. One API handles semantic encoding, importance scoring, intelligent decay, and learned retrieval ranking.

Code Complexity

[OLD] 300+ lines, 4 database connections, hardcoded weights

[NEW] 20 lines, 1 API connection, learned automatically

Maintenance

[OLD] Monitor 3 databases, manual tuning, write cleanup jobs

[NEW] Monitor one service, auto-tuning, intelligent decay

Quality Over Time

[OLD] Degrades. More data = more noise

[NEW] Improves. More data = smarter system

Developer Time

[OLD] 90% infrastructure, 10% agent logic

[NEW] 100% agent logic. Zero infrastructure.

WITHOUT ORBIT 300+ lines | **WITH ORBIT 20 lines**

```
        with self.db.connect() as conn:
            conn.execute("INSERT INTO interactions ...")

        redis_client.setex(
            f"recent_{user_id}", 86400,
            json.dumps([message, response])
        )

    def retrieve_context(self, user_id, message, limit=5):
        cached = redis_client.get(f"recent_{user_id}")
        embedding = openai_client.Embedding.create(...)
        vector_results = self.pinecone_index.query(...)

        with self.db.connect() as conn:
            db_results = conn.execute(...)

        return self._manually_rank_context(
            vector_results, db_results, cached
        )

    def _manually_rank_context(self, vectors, db, recent):
        # Hardcoded. Probably wrong. Never learns.
        ranked = []
        for item in recent:
            ranked.append({"importance": 0.9})
        for result in vectors:
            ranked.append({"importance": 0.7})
        for result in db:
            ranked.append({"importance": 0.5})
        return sorted(ranked, key=lambda x: x["importance"])[:5]
```

3 databases. Hardcoded weights. No learning. Python

REAL-WORLD SCENARIO

Building a coding chatbot. Two worlds compared.

A chatbot that helps users learn to code. It remembers skill level, topics covered, struggles, preferences, and code snippets. Watch how memory works without Orbit vs. with Orbit.

Day 1 Day 30 **Month 3**

1,000 users. System at scale.

WITHOUT ORBIT

01 Vector DB: 500K messages. Getting slow.

02 Response time: 2-5 seconds. Too much context to process.

03 Context window: 70% used. Cramped.

04 User satisfaction: 68% — declining.

05 Developer pain: $1000/month, 20+ hours/month tuning

RESULT Options: Hire ML engineer, redesign from scratch, or give up

DEV Decision: 'This is too hard. Never again.'

WITH ORBIT

01 Orbit storage: 150K memories. Only the important stuff.

02 Response time: 50-100ms. Clean context.

03 Context window: 20% used. Plenty of room.

04 User satisfaction: 87% — improving monthly.

05 Developer cost: $200/month. Zero hours tuning.

RESULT System automatically improved without any intervention

DEV Decision: 'I could build 10 more chatbots this easily.'

CAPABILITIES

What makes Orbit fundamentally different.

Not another vector database. Not a wrapper around embeddings. Orbit is a complete memory infrastructure with built-in intelligence that improves with every interaction.

--A

Semantic Encoding

LLM-powered understanding of every memory. No keyword matching. No regex. Pure understanding of meaning, intent, and relationships between concepts.

Every piece of information is processed through deep semantic analysis so your agent truly understands what it knows.

--B

Learned Importance

A neural network predicts what matters. Importance scores adapt over time based on real-world outcomes, not hardcoded guesses.

Orbit learns which memories actually improve agent responses. Importance is earned, not assigned.

--C

Adaptive Decay

Not all memories should last forever. Orbit learns optimal decay curves for different types of information, keeping memory lean.

Old beginner data automatically fades as users advance. Recent breakthroughs persist. The system knows the difference.

--D

Intelligent Retrieval

Ranked by actual usefulness in production, not just cosine similarity. Results get better the more your agent uses Orbit.

5 items from Orbit outperform 20 items from traditional retrieval. Less noise. More signal. Better answers.

--E

Continuous Learning

Every retrieval, every agent decision feeds back into the memory system. Orbit gets smarter without you writing a single new rule.

Feedback loops update importance models, decay curves, and ranking weights automatically. Ship once, improve forever.

--F

Full Observability

Know exactly what memories are being used, which ones lead to good responses, and what's just noise.

Answer 'Why did quality drop?' instantly. See what's helping, what isn't, and what should be remembered more aggressively.

AFTER 3 MONTHS

The numbers speak for themselves.

After 3 months with 1,000 users. Without Orbit: quality declining, costs rising, developer burned out. With Orbit: quality improving, costs flat, developer shipping features.

87%
User satisfaction
vs 68% traditional

+5%
Monthly improvement
vs -2% traditional

$200
Monthly cost
vs $1,000+ traditional

0 hrs
Dev time on memory
vs 20+ hrs traditional

METRIC	WITHOUT	WITH ORBIT
Setup Time	3+ days	15 minutes
Code Lines	300+	20
Databases to Manage	3+	0
Manual Tuning	Constant	Never
Quality at Month 1	70%	85%
Quality at Month 3	68% (declining)	88% (improving)
Context Window Waste	60%	20%
Monthly Cost	$1,000+	$200
Developer Hours/Month	20+	0
Scaling Difficulty	Hard	Easy
Learning Loop	None	Automatic
Improvement Signal	Blind	Transparent

DEVELOPER JOURNEY

Same effort. Different outcomes.

Week 1

WITHOUT ORBIT

"Let me set up Pinecone, Postgres, Redis..."

Spends 3 days on infrastructure. 1 day guessing at weights.

Chatbot works. Users: 'Okay, decent.'

WITH ORBIT

"Let me integrate Orbit..."

15 minutes setup. 15 minutes adding feedback collection.

Chatbot works. Users: 'Wow, it understands me!'

Month 1

WITHOUT ORBIT

"Why is quality degrading? Old data is interfering!"

Hours debugging. Writing cleanup jobs. Tweaking hardcoded values.

> User: 'Chatbot was good week 1, but getting worse'

WITH ORBIT

"Checks Orbit dashboard. Accuracy up 5%."

Does nothing. System improving on its own.

> User: 'This chatbot understands me better every day'

Month 3

WITHOUT ORBIT

"At a scaling limit. Database is slow. Quality plateaued."

Options: Hire ML engineer, pay for Vespa/Algolia, or redesign from scratch.

> Decision: 'This is too hard. Never again.'

WITH ORBIT

"Scaling effortlessly. 10,000 messages/day."

Context still clean. Quality improved to 87%.

> Decision: 'I could build 10 more chatbots this easily.'

THE BOTTOM LINE

WITHOUT ORBIT

90% of time on infrastructure.

Developer spends 90% of time managing infrastructure, 10% on chatbot logic. Quality degrades over time. Scaling is painful.

For the same effort:

1 chatbot that works okay

and degrades over time

WITH ORBIT

100% of time on what matters.

Developer spends 100% of time on chatbot logic. Quality improves over time. Scaling is automatic.

For the same effort:

10 chatbots that work great

and improve over time

That's why Orbit matters.

GET STARTED

Ready to build agents that remember?_

Stop building memory from scratch. Stop managing three databases. Start shipping intelligent agents that get smarter with every interaction.

GET EARLY ACCESS

REQUEST DEMO

or run: `pip install orbit-memory`

■ ORBIT

Memory infrastructure for AI developers.
Built for those who demand intelligence in
every layer of their systems.

hello@theorbit.dev

Documentation GitHub Discord Status Blog Careers