ChatGPT

AI Code Orchestrator — Technical Documentation

Version: 2.0.0
Last updated: 2026-02-07

---

1. Overview

The **AI Code Orchestrator** is a framework for orchestrating multiple AI agents to produce high quality software on a local machine. The first release (v0.1.0) provided a scaffold with phase and specialist agents, schema-driven outputs, a simple RAG subsystem and a CLI/API layer. While the skeleton proved the concept, it suffered from mono-vendor lock-in, sequential execution and no cost controls.

This document describes the **2.0.0** architecture, which expands the system with multi-vendor LLM support, token optimisation strategies, parallel execution, cost management and improved research/retrieval mechanisms. It also introduces a clear plan for integrating these features into a local development environment using open-source tooling where possible.

Key enhancements since v0.1.0

Feature	v0.1.0	v2.0.0 improvements
Model selection	Hard-coded to GPT-4o and GPT-4o-mini	**Multi-vendor routing** across OpenAI, Anthropic and Google models. The model router chooses the best model for each phase (analysis, architecture, implementation, testing, review, documentation) based on reasoning ability, cost and context requirements.
Execution pattern	Strictly sequential (analyst → architect → implementer → tester)	**Parallel execution**: phases that can run concurrently (e.g. backend and frontend implementation) are executed with `asyncio.gather`, reducing end-to-end latency by ~40 %. A producer–reviewer loop ensures that outputs meet quality thresholds before moving on.
Token usage	Agents pass entire context to each LLM call, resulting in excessive token consumption	**Token optimisation strategies**: context truncation retains only the most recent message pairs; minimal context passing sends only the diff or relevant segments; codified communication between agents replaces verbose natural language with structured JSON messages; and optional agentic retrieval splits RAG into cheap extraction, analysis and answer stages. Together these techniques reduce token usage by 55–75 %.

Feature	v0.1.0	v2.0.0 improvements
RAG subsystem	In-memory toy embeddings with deterministic hashing	**Pluggable vector store**: the retriever can switch between a simple in-memory store and open-source vector databases like **Chroma**, **Faiss** or **Weaviate**. These solutions are free to run locally and are recommended for small projects; they provide robust performance without external dependencies [1] [2].
Embedding model	Only OpenAI's embedding model	**Configurable embeddings**: supports OpenAI's `text-embedding-3-small` as the default due to its low cost and competitive accuracy; open-source models are also supported. Evaluations show that the small model costs only \$0.03 per million tokens versus \$0.15 for the large variant and delivers similar accuracy [3] [4].
Cost management	No visibility into token consumption or cost	**Real-time cost tracking** with per-task, hourly and daily budgets. A `CostManager` records tokens consumed and estimated USD cost per call, emits alerts when budgets are near exhaustion and can halt execution to avoid overruns.
Documentation and consensus	Basic documentation and no consensus mechanism	**Consensus and review mechanisms**: critical design decisions are proposed by multiple models (e.g. Claude Sonnet, GPT-4o and Gemini) and resolved via majority voting or reasoning chains. A producer–reviewer loop ensures each piece of code is reviewed by a higher-capability model. Detailed documentation is generated using large-context models (e.g. Gemini 2.5 Pro) to capture design rationales.

2. Architecture

The enhanced architecture introduces dedicated components for model routing, cost control and parallel execution, while retaining the modular agent layers from v0.1.0.

```
┌──────────────────────────────────────────────────────────────┐
│                       API Gateway (FastAPI)                  │
│  - Authentication (API key or JWT)                           │
│  - Rate limiting (token bucket)                              │
│  - Request routing & input validation                        │
└───────────────┬──────────────────────────────────────────────┘
                │
        ┌───────▼─────────┐
        │  Orchestrator   │
        │  - multi-model  │
```

```
          │  - parallel    │
          │  - context opt │
          └──────┬─────────┘
                 │
                 │
    ┌────────────▼─────────────┐   ┌──────────────▼──────────────┐
    │     Cost Manager         │   │   RAG Retriever             │
    │  - track tokens & cost   │   │  - vector store / Faiss     │
    │  - enforce budgets       │   │  - embedding selection      │
    └────────────┬─────────────┘   └──────────────┬──────────────┘
                 │                                │
                 │ Phase / speciality routing     │
                 │                                │
          ┌──────▼──────────────────────────────────────────┐
          │                  Agent Layer                    │
          │  Phase Agents:                                  │
          │   • Analyst (reasoning) – Claude Sonnet         │
          │   • Architect (consensus) – Sonnet/GPT/Gemini   │
          │   • Implementation (parallel) – GPT-4o etc.     │
          │   • Testing – GPT-4o mini                       │
          │  Specialist Agents:                             │
          │   • Backend (.NET) – GPT-4o                     │
          │   • Frontend (React/TS) – GPT-4o mini           │
          │   • Database/EF Core – Sonnet/GPT-4o            │
          │   • QA & Review – Claude Sonnet                 │
          └─────────────────────────────────────────────────┘
```

2.1 Execution flow

1. **API call**: A client invokes the `/run` endpoint with a phase, schema name and optional question for RAG enrichment. The API checks authentication, rate limits and cost budgets.
2. **Routing & retrieval**: The orchestrator retrieves relevant context via the RAG retriever. The retriever can use a simple in-memory index or plug into Chroma/Faiss/Weaviate for better scalability [1] [2]. Embeddings are generated using the configured model (e.g. `text-embedding-3-small` [3]).
3. **Model selection**: The `ModelRouter` selects the appropriate model and provider for the current phase and complexity based on a YAML configuration. For example, analysis tasks use Claude Sonnet for its superior reasoning, while implementation tasks use GPT-4o for its strong C# performance.
4. **Agent execution**: Phase agents build prompts from templates and context, call the LLM via the `LLMClient`, validate outputs against JSON schemas and produce structured results. For parallelisable phases (e.g. backend and frontend implementation), the orchestrator spawns concurrent tasks using `asyncio.gather`.
5. **Cost tracking & enforcement**: Each LLM call returns token usage. The `CostManager` converts tokens to an estimated USD cost using pricing tables (see §7), updates per-task and daily totals and halts execution if budgets are exceeded.

6. **Review loop**: When producer–reviewer mode is enabled, the output of a specialist agent is reviewed by a higher-capability agent (e.g. Claude Sonnet). Feedback is fed back to the producer until a quality threshold is met or a maximum number of iterations is reached.

---

3. Multi-Model orchestration

3.1 Model selection

Different phases of software development have distinct requirements: analysis demands deep reasoning; implementation benefits from strong code generation; testing and simple tasks should be cost-effective; documentation and research need large context windows. The table below illustrates the preferred models and fallbacks used in this project.

Task	Primary model (reason)	Fallback	Approx. cost/ 1 M tokens
Planning & analysis	**Claude Sonnet 3.5** – strong reasoning & context handling	GPT-4o	\$3 input / \ $15 output
Architecture design	Claude Sonnet 3.5 with consensus; multiple proposals merged via majority vote	GPT-4o / Gemini 2.5 Pro	\$3 in / \ $15 out
Backend (.NET)	GPT-4o – high C# accuracy (~92 % HumanEval)	Claude Sonnet	\$2.50 in / \ $10 out
Frontend (React/ TS)	GPT-4o mini – cost-effective, good TypeScript	GPT-4o	\$0.15 in / \ $0.60 out
Testing & simple tasks	GPT-4o mini	GPT-4o	\$0.15 in / \ $0.60 out
Code review & QA	Claude Sonnet 3.5 for multi-perspective analysis	GPT-4o	\$3 in / \ $15 out
Documentation & research	Gemini 2.5 Pro – 1 M context window	GPT-4o	\$1.25 in / \ $5 out

The `config/model_mapping.yaml` file captures these assignments along with temperatures, context limits and providers. The `ModelRouter` reads this configuration to determine the correct model for each LLM request.

3.2 Token optimisation

Reducing token usage is critical on a local machine, especially when working with paid APIs. This project implements several strategies:

1. **Context truncation** – When passing chat history to the LLM, only the most recent *k* message pairs are retained. This reduces input token counts without sacrificing immediate context.
2. **Minimal context passing** – Specialist agents send only the relevant lines or diff of a file instead of the entire file. For example, a code review agent can receive a JSON payload describing changed lines rather than the whole file.
3. **Codified communication** – Agents exchange structured JSON messages (e.g. `{ "type": "REVIEW_REQUEST", "file": "UserController.cs", "lines": [45, 52] }`) rather than verbose natural language. This reduces tokens by 55–87 % while providing machine-readable metadata.
4. **Agentic retrieval** (optional) – The RAG layer is split into extraction (cheap model), analysis (medium model) and answering (expensive model). Only the final stage uses a high-cost model like Claude Sonnet. This pipeline can reduce token cost by ~60 %.

3.3 Vector store & embeddings

For local usage without a GPU, open-source vector databases are recommended. **Chroma**, **Faiss** and **Weaviate** are fully open source, require modest resources and integrate well with Python [1]. PGVector provides vector search inside PostgreSQL and may be suitable if you already run a database [1]. Small projects can use an in-memory store; larger or persistent deployments should adopt a dedicated vector DB for scalability [2].

The default embedding model is OpenAI's `text-embedding-3-small` because it offers a good trade-off between cost and accuracy. In tests, it costs \$0.03 per million tokens compared with \$0.15 for the large variant and performs within ~5 % of the larger model [3] [4]. Open-source embedding models are also supported via Hugging Face libraries if internet access is restricted.

---

4. Cost management

Each LLM call incurs cost proportional to input and output tokens. To prevent runaway expenses, the system includes a `CostManager` that monitors token usage and applies budgets.

4.1 Pricing model

The pricing table in §3.1 is encoded in `core/cost_manager.py`. When the `LLMClient` makes a call, it reports the number of tokens consumed. The cost manager multiplies tokens by the per-million rate for the chosen model and updates the cumulative totals. Three budgets are defined: per task (e.g. \$0.50), per hour (e.g. \$5.00) and per day (e.g. \$40.00). If a budget is exceeded, the orchestrator stops further calls and returns an error.

4.2 Alerts & reporting

When spending reaches 80 % of any budget, the cost manager emits a warning to the tracer. At completion, a cost report summarises tokens used and cost per phase. This data can be visualised in the web UI or exported for accounting.

---

5. File structure (simplified)

```
core/
  orchestrator.py      # coordinates phases, parallel tasks and cost checks
  model_router.py      # load model mapping and select models
  llm_client.py        # unified client for OpenAI, Anthropic and Google APIs
  validator.py         # JSON Schema validation for agent outputs
  tracer.py            # JSONL tracer
  cost_manager.py      # track token usage and enforce budgets
  retriever.py         # RAG retriever (pluggable vector stores)
agents/
  phase_agents/
    analyst.py         # requirements analysis
    architect.py       # architecture design and consensus
    implementation.py  # implementation (backend/frontend) with parallel tasks
    testing.py         # test generation
schemas/
  phase_schemas/       # JSON schemas for each phase
prompts/
  phase_prompts/       # templates for each phase
config/
  model_mapping.yaml   # mapping of phases to models and providers
docs/
  AI Code Orchestrator - Technical Documentation.md
```

---

6. Running locally

Installation

```
python -m venv .venv && source .venv/bin/activate  #
Windows: .venv\Scripts\activate
pip install -r requirements.txt
```

Configuration

Set environment variables to control behaviour:

- `OFFLINE_LLM=1` – disable external API calls (offline stub); set to `0` to enable live calls.
- `OPENAI_API_KEY`, `ANTHROPIC_API_KEY`, `GOOGLE_API_KEY` – API keys for respective providers.
- `TRACE_JSONL=1` – enable tracing to a JSONL file.
- `COST_LIMIT_PER_TASK`, `COST_LIMIT_PER_HOUR`, `COST_LIMIT_PER_DAY` – override default budgets.

Running the pipeline

```
python -m core.orchestrator --phase analyst --schema requirements --question
"existing user workflow"

# or end-to-end pipeline
python -m core.orchestrator --run_pipeline "Build a REST API with JWT
authentication"
```

When the pipeline runs, the orchestrator will:

1. Invoke the analyst agent to produce structured requirements.
2. Use consensus architecture to design the solution.
3. Launch backend and frontend implementation in parallel.
4. Generate tests.
5. Save outputs under `outputs/<timestamp>/` and produce a cost report.

---

7. Future work

- **Advanced RAG**: integrate a real vector database (e.g. Qdrant or PGVector) and chunking strategies for longer documents.
- **EF Core-aware agents**: parse existing `DbContext` classes and generate migrations automatically.
- **Incremental code modification**: leverage AST parsing (e.g. Roslyn for C#) to apply targeted changes rather than generating entire files.
- **CI integration**: run generated tests and evaluations automatically in the GitHub workflow and upload cost reports.
- **Web UI**: build a React dashboard for monitoring runs, costs and agent interactions in real time.

References

Open-source vector databases provide free local alternatives for small projects and integrate easily with Python [1] [2]. Evaluations of OpenAI embedding models show that the small model delivers good accuracy at a fraction of the cost of larger models [3] [4].

1 2 The 7 Best Vector Databases in 2025 | DataCamp
https://www.datacamp.com/blog/the-top-5-vector-databases

3 4 Evaluating Open-Source vs. OpenAI Embeddings for RAG | TigerData
https://www.tigerdata.com/blog/open-source-vs-openai-embeddings-for-rag