Deep Research Into Quality Control for Agentic Coding

Here's a concrete blueprint for an **agentic autopilot** that can plan, build, test, review, and continuously refine an entire program—editing its own roadmap while staying laser-aligned to the original purpose, and doing it on an M1 Mac mini without melting your fans.

---

0) First principles (guardrails that make this work)

- **Goal-locked, plan-flexible.** The *purpose* is fixed; the *roadmap* is mutable. Encode the purpose as invariants, acceptance criteria, and non-functional constraints (NFRs). Let the roadmap be a living, ranked backlog the autopilot can rewrite—but every change must cite how it improves purpose/NFR satisfaction.
- **Role separation + hard gates.** Use small, specialized agents (Planner, Coder, Tester, Reviewer, Roadmap-Editor) with pass/fail gates. No role can "skip ahead" without producing required artifacts (Cited Plan, tests, review notes, etc.).
- **Evidence ledger.** Every decision produces evidence (citations, diffs, test results). Merges only happen with green gates and linked evidence. If it's not in the ledger, it didn't happen.
- **Live-fire over compile-only.** Always run the program (or a realistic harness), not just lints. Add property-based tests + a tiny mutation budget per PR.
- **Tight compute loop.** Do the minimum necessary work each iteration: diff-based indexing, small candidate sets, cached retrieval, small batches.

---

1) System architecture

Control plane (one small orchestrator process):

- **Orchestrator** (state machine) runs the macro-loop and enforces gates.
- **Task Router** picks the next work item (from roadmap/backlog).
- **Evidence Store** (SQLite) persists artifacts: plans, diffs, test logs, votes.
- **KB / Retrieval**: FAISS (vectors) + SQLite/Tantivy (lexical), populated by a **Diff Indexer**.

Worker agents (can be the same model with different SOPs):

- **Planner** → decomposes goal/epic into tasks; emits *Cited Plan*.
- **Coder** → implements changes; emits code + verify.sh/py.
- **Tester** → writes example tests + property-based tests + symmetry cases; runs tests; reports coverage & mutation score.
- **Reviewer** → static/security review, round-trip requirement check,

objections list.

- **Roadmap-Editor** → proposes roadmap edits (reprioritize, split/merge tasks) with proof (metrics delta, evidence).

Tools (on M1):

- Python runtime, pytest, **Hypothesis** (PBT), mutmut (or cosmic-ray) with a **small mutant budget** (e.g., 20/PR).
- Linters: ruff, mypy (Py) or ESLint/TS; security: bandit/Semgrep.
- Git (local), Makefile, GitHub Actions (optional), shell access.

---

2) Data & memory model

ProjectPurpose.yml (the North Star):

```
purpose: "High-quality X that does Y for Z users"
acceptance_criteria:
  - ...
non_functional_requirements:
  - reliability >= 99.9% simulated
  - latency_p95 <= 120ms on test workload
  - maintainability: docstrings for public APIs, cyclomatic complexity <= N
guardrails:
  - prohibited_patterns: [...]
  - required_patterns: [...]
```

Roadmap.json (editable by the autopilot):

```
{
  "epics": [
    {"id":"E1","name":"Core API","prio":1,"status":"in_progress","value":0.35},
    ...
  ],
  "tasks": [
    {"id":"T14","epic":"E1","name":"Add exponential backoff","status":"todo",
     "deps":["T09"],"risk":"med","value":0.12,"evidence":[]}
  ]
}
```

Evidence ledger (SQLite tables): decisions, artifacts, test_runs, reviews, votes, mutations, all keyed by git SHA and task id.

KB/Index:

- **Chunkers:** AST-level for code; heading+sentences for docs/ADRs; per-test for /tests; per-comment for PR/issue text.
- **Metadata:** repo, path, commit, symbol, kind, headings, ADR ids, issue/pr ids, owner, timestamps.
- **Hybrid search:** BM25 ∪ FAISS → cross-encoder rerank → grouped by intent (ADRs, closest symbols, tests, incidents).

---

3) Macro loop (the autopilot state machine)

```
LOOP
  Sense:
    - Pull repo diff, update KB (changed files only)
    - Compute health snapshot: test pass %, coverage, mutation score, open objections
  Decide:
    - Select next task (value/risk/blocked?); or let Roadmap-Editor propose reordering
    - If roadmap change proposed, require proof (metrics simulation or constraints improvement) and pass a vote
  Plan (Planner):
    - Produce Cited Plan:
        • goal + acceptance criteria
        • impacted symbols/modules
        • related ADRs/specs/tests (top-k citations)
        • plan-of-attack + verification plan (invariants, symmetry cases, PBT outline)
    - Gate: must cite ADR + at least 1 negative test/incident + target invariants
  Act (Coder):
    - Implement; create or update verify.sh/py; write docstrings
    - Small commits with meaningful messages
  Check (Tester):
    - Generate example tests from acceptance criteria
    - Write 1+ property-based test (PBT) for a named invariant
    - Instantiate symmetry-guided cases (SGAT) for that component
    - Run unit+integration; compute coverage
    - Run a small mutation budget; compute mutation score
    - Gate: tests green AND coverage threshold AND mutation score >= floor
  Review (Reviewer):
    - Run static/security tools; summarize issues
    - Round-trip review: "From code, reconstruct requirement; compare"
    - Gate: zero unresolved objections
  PR & Merge:
    - Create PR with template (Cited Plan, evidence table, risks, rollbacks)
    - If all gates green, merge to main; else loop back to previous phase
  Learn:
    - Update Roadmap.task value/risk using evidence (e.g., defects found)
    - Update heuristics (where mutants died → future testing focus)
END
```

Fail-fast gates (non-negotiable):

- Missing *Cited Plan* with ADR/test citations → block.
- No PBT or no symmetry cases → block.
- Mutation score regression vs baseline → block.
- Reviewer objections outstanding → block.

---

4) How the roadmap can self-edit without drifting off purpose

Roadmap-Editor agent policy:

- May: split a task, merge duplicates, reprioritize, add discovery tasks.
- Must: justify with **evidence**: improved value/effort ratio, risk reduction, or constraint satisfaction (e.g., shaving 40ms p95).
- Must not: alter **purpose** or acceptance criteria. Any suggestion impacting purpose must be rejected by the orchestrator.

Change process:

1. Editor proposes a patch to Roadmap.json with deltas.
2. **Impact preview:** run "semantic impact set" (what else co-changes?), recalc dependency risk.
3. **Vote:** Planner + Reviewer + Orchestrator (simple majority) accept/reject.
4. On accept: commit Roadmap patch; post rationale in ledger.

This gives you *adaptive planning* without goal drift.

---

5) Quality: the four levers you wire in from day one

1. **Cited Plan** → forces intent alignment (Planner must pull ADRs/specs/tests/incidents).
2. **PBT + SGAT** → forces tests that fight back (Tester must encode invariants and symmetry).
3. **Mutation budget** → forces fault-finding beyond coverage.
4. **Round-trip review + security/static** → forces conceptual and non-functional correctness.

All four are cheap enough for an M1 when budgeted (e.g., ≤20 mutants;

Hypothesis with small examples; cached retrieval).

---

6) Minimal repo scaffold

```
.autopilot/
  orchestrator.py          # state machine
  sop/                     # role SOPs (prompts with checklists)
    planner.md
    coder.md
    tester.md
    reviewer.md
    roadmap_editor.md
  gates.yml                # thresholds (coverage, mutation floor, lint severity)
  templates/
    pr.md
    cited_plan.md
    review_checklist.md
  tools/
    symmetry.py            # SGAT helpers (inversion, idempotence, etc.)
    metrics.py             # read coverage.xml, junit.xml, mutation.json
    kb_indexer.py          # AST/doc chunker, FAISS/SQLite loaders
  kb/
    faiss.index
    metadata.sqlite
  state/
    ledger.sqlite
ProjectPurpose.yml
Roadmap.json
Makefile
.github/workflows/ci.yml
```

---

7) SOP excerpts (copy/paste and tweak)

Planner SOP (cited plan)

- Inputs: ProjectPurpose.yml, Roadmap task, KB search results.
- Output must contain:
- Acceptance criteria restated
- Impacted symbols/modules (paths, lines)
- **Citations:** ≥1 ADR/spec, ≥1 existing test or incident

- Verification plan: list invariants; symmetry cases; PBT outline
- Exit: fail if any required citation or invariant is missing

Tester SOP (tests that fight back)

- Write example tests from acceptance criteria.
- Write ≥1 Hypothesis test encoding a named invariant.
- Instantiate symmetry cases from tools/symmetry.py (e.g., encode/decode; noop; commutativity/idempotence where applicable).
- Run tests + coverage; run mutation budget (≤20); produce test_report.json with coverage%, mutation score, failing cases (if any).
- Exit: fail if mutation score < floor or no PBT/symmetry present.

Reviewer SOP (round-trip + security)

- Run ruff/mypy/bandit (or ESLint/Semgrep). Flag issues.
- Round-trip: summarize what the code **does**; diff against acceptance criteria; list mismatches.
- Approve only if zero unresolved issues and round-trip aligns.

---

8) CI wiring (lean but strict)

Makefile

```
index:
\tpython .autopilot/tools/kb_indexer.py --diff
test:
\tpytest -q --maxfail=1 --disable-warnings
pbt:
\tpytest -q -k "property_"   # tag PBTs
mut:
\tmutmut run --paths-to-mutate src/ --tests-dir tests/ --use-coverage
lint:
\truff check src tests && mypy src && bandit -r src
verify:
\tbash ./verify.sh || python verify.py
gates:
\tpython .autopilot/tools/metrics.py --enforce .autopilot/gates.yml
```

.github/workflows/ci.yml (sketch)

- Step 1: make index
- Step 2: make test pbt verify
- Step 3: make mut (cap runtime via timeout)
- Step 4: make lint

- Step 5: make gates (fail on thresholds / missing artifacts)
- Step 6: Autopilot posts review (or blocks merge)

---

9) Orchestrator skeleton (pseudo-Python)

```
while True:
   kb.update_diff()
   health = snapshot_health()
   task = select_task(roadmap, health)  # or let editor propose patch + vote

   plan = Planner.cited_plan(task, kb, purpose)
   gate(plan.ok, "Missing citations or invariants")

   diff = Coder×implement(plan)
   run("git add -A && git commit -m 'WIP: {}'".format(task.id))

   tr = Tester.run_all(plan, diff)  # unit+integration+PBT+SGAT+mutation
   gate(tr.green and tr.mutation >= floor and tr.pbt and tr.sgat, "QA failed")

   rv = Reviewer.review(diff, plan, tr)
   gate(rv.ok and not rv.objections, "Review failed")

   pr = create_pr(plan, tr, rv)
   if all_gates_green(tr, rv):
      merge(pr)
      update_roadmap_success(task, tr)
   else:
      log_and_refine(task, tr, rv)
```

---

10) How it avoids your current pitfalls

- **Superficial tests:** PBT + symmetry + mutation **force** robust tests; CI blocks if absent or weak.
- **Skipping steps:** Role gates with artifact checks prevent “fast-forward”; missing Cited Plan or objections ⇒ loop back.
- **No coherence:** Hybrid retrieval (symbol-aware) and **semantic impact sets** demand that related modules/tests change together; planner must cite ADRs/specs.
- **No live-fire:** Every task must ship a verify.sh/py that runs a realistic scenario; CI runs it on every PR.

---

11) Day-by-day bring-up (1–3 days to MVP)

Day 1

- Add ProjectPurpose.yml, initial Roadmap.json.
- Drop .autopilot/ scaffold; implement KB indexer (diff-only), FAISS/SQLite.
- Write SOPs and gates; wire Makefile + CI minimal ("test + lint + verify").

Day 2

- Implement Planner, Coder, Tester, Reviewer wrappers (simple prompts).
- Add symmetry.py helpers; tag at least one PBT in your codebase.
- Add mutmut with a 10–20 mutant budget targeting touched files.

Day 3

- Enforce PR template & gates. Turn on block-on-red.
- Add Roadmap-Editor with vote and evidence requirements.
- Start collecting metrics (coverage, mutation score, objection count, defect escape).

You'll feel the upgrade immediately: agents will slow down just enough to be *correct*, and PRs will come with real evidence.

---

12) Metrics that actually matter

- **Per-PR:** coverage %, mutation score, #symmetry tests, #PBTs, #review objections (should trend to 0), cycle time.
- **Per-epic:** defect escape rate (issues after merge), p95 latency under verify workload, coherent co-changes (ratio of impacted siblings touched vs expected).
- **System:** retrieval nDCG@k on a tiny gold set of "must-find" ADRs/tests; build flake rate; CI wall time.

Final note

You don't need a massive model zoo or swarm to get "elegant, efficient, high-quality" autonomy. You need: **goal-locked planning, role gates, tests that fight back, a tiny mutation budget, and a ledger of evidence.** Start with the scaffold above, keep the budgets small for your M1, and let the autopilot *earn* every merge.

Semantic search is about retrieving things by meaning, not just by matching the same words. In a codebase that sprawls across source files, tests, docs, issues, PRs, ADRs, configs, and dashboards, “meaning-first” retrieval is the difference between an agent (or a human) finding the right constraint/spec/example in seconds vs. skating past it and producing brittle work. Below is a practical, opinionated blueprint for using semantic search to strengthen quality during a task, preserve coherence across a project, and keep agents honest.

What it is (and why it matters in a repo)

Lexical search (grep/BM25) matches tokens; it’s fast and exact but misses synonyms, paraphrases, and cross-language intent.
Semantic search embeds queries and documents into a vector space so “create a streaming retry with backoff” can surface a code sample named resilient_http.py, an ADR about exponential backoff, a PR discussion on idempotency, and a test titled “retries on 5xx”—even if none use your exact phrasing.

Concrete wins in complex repos

- **Intent alignment:** Retrieve the acceptance criteria, ADR, and non-functional constraints that should govern the change—before coding.
- **Cross-artifact recall:** Link code to design docs, incidents, runbooks, and SLOs so fixes respect real production behavior.
- **Refactor safety:** Find semantically similar usages/APIs to change together (e.g., “soft delete” semantics across services).
- **Test integrity:** Pull closest negative/edge-case examples and incident postmortems, not just “happy path” unit tests.
- **Onboarding & memory:** Agents stop “forgetting” prior decisions; humans stop copy-pasting anti-patterns.

Design principles that keep quality high

1. **Right granularity:** Index at multiple levels: file → section → **symbol/AST node** → code block → test case → doc paragraph → PR comment. Symbol-level recall prevents overlong chunks and hallucinated context.
2. **Preserve context:** Store **structural metadata** (repo, package, path, commit, symbol kind, parents/children, owning team, ADR id, Jira id, environment) and a **breadcrumbs string** you can display in UX.
3. **Freshness by diff:** Re-index only changed chunks on each commit/merge; never rebuild the world.
4. **Hybrid first:** Start with BM25/keyword to guarantee exact matches,

union with ANN vector results, **rerank** with a cross-encoder. This keeps precision without losing recall.

5. **Type-aware chunking:** Different rules for code, tests, docs, configs, issues. Chunk by **AST node** for code, **headings + sentences** for docs, **YAML blocks** for configs.
6. **Guardrails for agents:** Make retrieval **mandatory** at key workflow stages ("strategize → plan → implement → verify → review"). If the agent can't cite top-k governing artifacts, it's not allowed to proceed.
7. **Evaluate rigorously:** Track nDCG@k, Recall@k, and **Task Success under Retrieval** (TSuR)—does the right context appear in the final plan/PR/tests?

Implementation blueprint (fits an M1 Mac Mini)

1) Sources & schema (one index, many types)

Index these **uniformly**:

- /src code symbols (functions, classes, endpoints), comments, docstrings
- /tests cases + names + parametrizations + fixtures
- /docs (Markdown/MDX), READMEs, design specs, ADRs
- PR descriptions & review comments; Issues & RFCs
- /config (YAML/TOML/JSON), CI pipelines, infra manifests
- Runbooks, incident postmortems, SLO/SLI docs, dashboards (exported text)

Metadata (must-have):

repo, path, commit_sha, artifact_type, language, symbol, symbol_kind, hierarchy (module>class>method), heading, tags, adr_id, issue_id, pr_id, owner, service_name, env, created_at, updated_at

2) Chunking (per artifact type)

- **Code:** Parse AST; chunk per symbol (≤ 300–600 tokens). Attach parent signature and leading docstring. Keep imports in parent chunk, not every child.
- **Docs/ADRs:** Split by heading; within sections, by 6–8 sentences or ~800–1200 chars; keep the heading chain (H1>H2>H3) as context.
- **Tests:** Each it/describe/test_*/pytest-function is one chunk; include parametrization + markers; tag with the production symbol(s) it targets (you can infer with static import analysis or naming heuristics).
- **Issues/PRs:** Paragraph-level chunks; attach labels, files touched, and linked issues/ADRs; pull reviewer comments as separate chunks with "stance" (approve/block) if you can infer.

3) Embeddings & storage

- **Embeddings:** Use a single strong general-purpose model for text and code, or a dual setup: one for **code** (e.g., a code-specialized sentence transformer) and one for **natural language**. If you want fully local on M1, use a **sentence-transformers** family model (e.g., bge / e5 / all-MiniLM variants) plus quantization. If you're OK with a hosted API, use any high-quality modern embedding with long context and good multilingual/code performance.
- **Vector store:** FAISS (Flat + HNSW/IVF) locally is fine on M1. If you want persistence & filters, use SQLite + FAISS or a lightweight server (Qdrant/Weaviate) Docker'd with ARM builds.
- **Lexical store:** tantivy (via pytantivy) or whoosh for pure-Python; or just ripgrep for CLI-users. You'll use it in hybrid union.

4) Query pipeline (agent + human)

Inference-time steps:

1. **Query understanding:** Detect intent ("Where is exponential backoff defined?" → code + ADR). Expand with synonyms via short LLM call if needed.
2. **Candidate generation:**
   - Lexical: BM25 top 100 (fast).
   - ANN: vector top 200 with filters (e.g., language=python, service=payments, artifact_type in [adr, test, code]).
3. **Rerank:** Cross-encoder (bi-encoder embeddings retrieve; cross-encoder re-scores top 300 to top 20). This massively improves precision@k.
4. **Context composer:** Group by **intent**: "Governing decisions (ADRs)", "Closest code symbols", "Closest tests", "Recent PRs touching same symbol", "Incidents touching same code path".
5. **Answer or cite:** For humans, show a slim UI: left column = facets/filters; right = grouped results with breadcrumbs and **inline diffs**. For agents, return **citations** + **canonical snippets** and require a **Cited Plan** section in the next step.

5) Integrations that improve quality

- **CLI:** repoq "add exponential backoff to ingest" --types adr,code,tests --service ingest → prints grouped results. Add --open to jump to file:line.
- **Editor:** VSCode command "Find governing artifacts" binds to current symbol → shows ADRs/tests/PRs.
- **Agent loop (hard gate):**
- **PLAN phase:** Agent must include Citations: with k≥5 covering ADR, test, code, incident.
- **IMPLEMENT:** Before edits, run "semantic impact set" → retrieve semantically similar symbols across repo; if plan doesn't include them, fail with TODOs.
- **VERIFY:** Generate tests seeded by retrieved **negative examples** and

incident patterns, not just stubs.

- **REVIEW/PR:** Auto-fill PR template with "Related artifacts" from retrieval; block merge if empty or low-quality.
- **CI hook:** On PR, compute embedding of diff context; retrieve conflicting specs/constraints; post a **bot review comment**: "Spec X contradicts change in Y" with citations.

6) Freshness & compute

- **Pre-commit (dev):** On staged changes, re-chunk only touched files; update embeddings for changed chunks; post to local FAISS.
- **CI (main):** On merge, run a thin indexer job; cache embeddings by content hash (sha1(content)).
- **Sharding:** If repo is huge, shard by service; load the shard index on demand.
- **Resource caps on M1:** Batch embed (e.g., 32–64 items), use float16 or 8-bit quantization; limit rerank to top 200; keep FAISS HNSW efSearch modest (64–128).

Using semantic search to enforce tension and stop shortcuts

You described agents skipping steps and writing tests designed to pass. Bake semantic search into **process control**:

- **Cited acceptance criteria:** Plan must cite the acceptance criteria, related ADR, and at least one negative test from history. Missing? **Block**.
- **Counterfactual test mining:** Retrieve incidents/PRs mentioning failures and auto-synthesize **regression tests** first; code must make these pass.
- **Cross-PR drift:** Retrieve historical PRs that changed the same symbol or interface; if a previous decision would be undone, require a "Change justification" block linking the old ADR/PR.
- **Production traces:** If you can export trace/span names/messages as text, index them. Retrieval then seeds **property-based tests** with real parameter distributions.

Evaluation plan (so you know it's working)

- **Offline IR metrics:** nDCG@10, Recall@20 using a small but strong labeled set (pair queries with must-contain artifacts).
- **Task-level metrics:** Given a set of tasks (e.g., "add exponential backoff"), does the agent **include the governing ADR and at least one negative test** in the plan? Track pass/fail.
- **Human study:** Time-to-first-useful-artifact; PR defect rate; test

flakiness delta; post-merge incident rate for files touched by agentic changes vs. baseline.

- **Drift monitoring:** Embed calibration checks monthly—sample queries, compute embedding similarity to gold labels; alert on drop.

Pitfalls (and how to dodge them)

- **Over-chunking:** Too small → no context; too big → irrelevant padding. Fix with **multi-scale chunks** and keep parent headings/symbol chain in metadata.
- **Embedding drift:** Mixing models over time degrades similarity. Pin model versions; store embedding_model_version in metadata; re-index only when you intend to.
- **No lexical backstop:** Pure vector search misses exact symbol names and error codes. Always hybridize.
- **Secret leakage:** Never index .env, private keys, tokens. Maintain a blocklist + secret scanner in the indexer.
- **Non-deterministic agent prompts:** Retrieval helps, but require agents to **quote** retrieved snippets and **link** to file:line; enforce with PR template checks.
- **Rerank cost:** Cross-encoders are expensive; cap candidates; cache query→results for 15–60 minutes per branch.

A minimal, practical stack to ship this week

- **Indexer:** Python; tree-sitter for AST; markdown-it for docs; gitpython for diffs.
- **Embeddings:**
- **Local-only:** sentence-transformers (e.g., e5-base or bge-base), quantized on M1.
- **Hosted:** Any modern embedding API with code + text strength.
- **Stores:** FAISS (HNSW) for vectors; SQLite + sqlite-vec or Tantivy for lexical; both filtered by metadata.
- **Reranker:** Cross-encoder from sentence-transformers (run locally in fp16) or a hosted rerank API.
- **Service:** FastAPI app exposing /search and /semantic-impact.
- **CLI/Editor:** A thin CLI; VSCode extension calling the FastAPI endpoints.
- **CI:** GitHub Action that runs indexer-on-diff and posts review comments with citations.

Skeletons you can adapt quickly

Index-time (Python):

```
# pseudo: build hybrid indices with diff-based updates
from pathlib import Path
from my_chunks import chunk_repo   # your AST/Markdown/YAML chunker
from my_embed import embed_batch   # your embedding wrapper
from my_store import VecStore, LexStore

def index_repo(repo_root, changed_paths=None):
    chunks = chunk_repo(repo_root, changed_paths)  # yield dicts {id, text, meta}
    texts = [c["text"] for c in chunks]
    embs = embed_batch(texts, batch_size=32)
    VecStore.upsert([(c["id"], e, c["meta"]) for c, e in zip(chunks, embs)])
    LexStore.upsert([(c["id"], c["text"], c["meta"]) for c in chunks])
```

Query-time (FastAPI-ish):

```
def hybrid_search(q, k=20, filters=None):
    lex = LexStore.search(q, top_k=100, filters=filters)
    ann = VecStore.search(q, top_k=200, filters=filters)  # ANN vectors
    cand = dedupe_merge(lex + ann, limit=300)
    reranked = cross_encoder_rerank(q, cand)[:k]
    return group_by_intent(reranked)  # code/tests/ADRs/incidents
```

Agent hard gate (plan phase):

```
results = hybrid_search("constraints for changing retry policy in ingest", k=20,
                        filters={"service":"ingest"})
required = have_types(results, ["adr","test","code"])
if not required:
    fail("Plan must cite ADR+test+code. Rerun retrieval.")
```

How this plugs into your "guaranteed quality" workflow

- **During a task:** The agent cannot move from "Plan" to "Implement" without a **Cited Plan** surfaced by semantic search.
- **Across the project:** Coherence is maintained by **semantic-impact sets** (find siblings to change) and **decision lineage** (ADRs/PRs surfaced automatically).
- **Context management:** Retrieval is **type-aware** and **symbol-scoped**, so long context windows aren't flooded with irrelevant text; you pass only the top grouped snippets with breadcrumbs.

- **Real testing vs. fake tests:** By mining incidents, failing tests, and negative examples semantically, generated tests reflect real-world behavior instead of trivial compile checks.
- **No compute drama on M1:** Local FAISS + light models + batched reranking keep latency modest; CI does heavier lifting on merges.

If you want, I can tailor this into a concrete checklist and a tiny reference implementation (indexer + FastAPI + VSCode command) aligned to your repo layout and the MCP/autopilot flow you're building.

Autonomous Agentic Coding & Orchestration for High-Quality Code

Introduction & Problem Statement

Autonomous **agentic coding** refers to using AI agents (often large language models) to handle software development tasks end-to-end – from planning and coding to testing and maintenance – with minimal human intervention . In theory, such agents can **plan, execute, validate, and iteratively improve code** on their own , accelerating development and ensuring consistency. However, achieving **reliable, high-quality code** through an agentic pipeline is challenging. Naïve implementations tend to rush through steps or skip rigorous checks, resulting in superficial outputs. For example, agents might generate unit tests that are trivial "happy path" cases designed solely to make the current code pass, rather than truly validating the intended functionality. Agents may also treat each development phase in isolation – doing the bare minimum to move to the next step without *tension* or cross-checking against previous stages. The outcome is often code that technically passes simple tests but fails in real scenarios, or misaligns with the original requirements. To guarantee quality, we need a more **structured, multi-step workflow** that enforces diligence at each stage and facilitates feedback loops. This answer outlines how to orchestrate a 10+ step autonomous coding process with specialized agent roles, rigorous testing, and hierarchical project management to consistently produce high-quality code.

Challenges in Naïve AI Coding Pipelines

Basic large-model coding pipelines typically follow a linear "send prompt -> get code -> get tests -> done" sequence. Such **Sequential One-Agent Pipelines (SOP)** suffer from well-documented limitations. Each stage assumes the previous output was correct and focuses narrowly on its own task, with **minimal cross-stage verification or error back-propagation** . There is no mechanism to reconcile misunderstandings or rectify mistakes from earlier phases, so early misinterpretations of requirements (e.g. missing an implicit constraint or edge case) propagate unchecked through later stages . Importantly, testing is often left as a final step and done by the same agent that wrote the code, which **lacks truly**

adversarial scrutiny . As a result, the agent might produce tests that simply confirm the implementation's obvious behavior (ensuring the tests pass easily) instead of challenging the solution. This one-way, **"no tension" workflow** tends to yield suboptimal results: studies have found that such pipelines often have high failure rates on non-trivial problems . In competitive coding benchmarks, for example, lack of coordinated adversarial testing and semantic drift between stages led to many wrong answers and inefficient solutions . In summary, a naive agent pipeline is prone to:

- **Semantic drift:** The intended meaning of requirements gets lost as each step loosely interprets the prior step, without verification .
- **Trivial testing:** Quality assurance is weak – tests are too easy or done too late to catch design/implementation flaws .
- **No feedback loops:** Errors discovered in testing (if any) aren't fed back to re-plan or refactor earlier components .
- **Fast-but-flawed output:** Agents may skip thorough analysis in favor of finishing tasks quickly, since there's no penalty for shallow work until the very end.

Overcoming these issues requires rethinking the pipeline to enforce **accountability and collaboration** among multiple agent roles, rather than a single-pass handoff.

Hierarchical Multi-Agent Orchestration

A proven approach to improve reliability is to adopt a **hierarchical multi-agent system (HMAS)**, where different AI agents assume specialized roles (planner, coder, tester, etc.) under a coordinated orchestration. Instead of one agent doing everything sequentially, we have an **orchestrator (leader agent)** that divides the work and several **specialist agents** that handle different phases of development . This is analogous to a human software team with a project manager, developers, and QA testers – each focusing on their expertise but working in concert. Research frameworks like MetaGPT and ChatDev demonstrate the effectiveness of this pattern:

- **Dedicated Roles & SOPs:** *MetaGPT* (Hong et al. 2023) models an AI software team by hard-coding roles such as **Project Manager, Architect, Coder, and Tester**, and embedding human-like Standard Operating Procedures into each role's prompt . This means, for example, the "Tester" agent is instructed with detailed procedures on how to independently verify features, and the "Architect" has guidelines for reviewing designs. Crucially, the roles are designed to **cross-check each other's outputs**, catching errors a single agent might miss . This built-in peer review mechanism addresses the lack of tension – e.g. the tester agent will scrutinize the coder agent's work, rather than trusting it blindly.
- **Structured Phases:** *ChatDev* (Qian et al. 2023) similarly divides software development into sequential phases (design, coding, testing) with specialized agent roles like **Requirements Analyst, Programmer, Reviewer, and**

Test Engineer. The agents engage in multi-turn dialogue during each phase to refine the outcome . For instance, during the testing phase ChatDev has a **code reviewer agent (static analysis)** and a **tester agent (dynamic testing)** who communicate to validate the code thoroughly . This led to improved completeness, executability, and consistency with requirements compared to a single-pass approach . In essence, ChatDev mimics a waterfall model pipeline using LLM agents, ensuring that each stage (from design to code to test) is handled by the appropriate expert and reviewed in natural language before proceeding .

- **Three-Tier Hierarchy:** Many architectures implement a three-tier hierarchy: **Strategy → Planning → Execution**. The top layer is a **strategy or leader agent** that interprets the overall goal and decides high-level priorities/order of work . The second layer is **planning agents** that take the strategy and produce concrete task breakdowns or design plans for the workers. The third layer consists of **execution agents** (workers) that perform the actual tasks – e.g. writing a code function, running a test, calling an API, etc . This hierarchy sharpens focus: the leader ensures global coherence, planners manage intermediate objectives, and workers concentrate on specialized subtasks. Empirical evidence shows hierarchical coordination yields better accuracy on complex tasks (and often better efficiency) than a flat single-agent approach . The hierarchy prevents the chaos of a "swarm" of agents by giving clear structure and oversight, much like having team leads in larger projects.

A **central orchestrator** can be implemented to coordinate this hierarchy. The orchestrator agent receives the user/project goal and assigns work to the appropriate specialist in sequence, integrating their results. It handles **task decomposition, sequencing, and result integration** . Notably, some advanced orchestrators use dynamic policies to optimize the workflow: for example, the *Puppeteer* framework uses a reinforcement-learned orchestrator that **decides which agent to invoke next and can skip unnecessary steps**, thereby cutting computation cost **while maintaining quality** . In a resource-constrained environment (like running on a local M1 Mac mini), such dynamic orchestration is valuable – it avoids wasting cycles on redundant subtasks, and can prune branches that add little value . The trade-off is that the orchestrator must be sophisticated enough to recognize when a step is safe to skip; otherwise, every critical quality-check step should be kept in place to avoid backsliding on reliability. In practice, a conservative approach is to **always run the key quality assurance steps** (planning, testing, code review) and only skip truly optional or repetitive ones.

Crucially, each agent should operate with **explicit instructions and context** relevant to its role. Effective agentic coding systems often include a *manifest* or structured prompt for each role that outlines its responsibilities, tools, and the context of the project . For example, a "Claude.md" or similar manifest might specify how the coding agent should format code, what standards to follow, and how to use memory or tools, while the testing agent's manifest might include instructions to generate edge-case inputs and expected outputs . Standardizing these operating procedures across the agents leads to more reliable outputs, as found in studies – clear, **actionable prompts and documentation for each agent correlate with better consistency and success rates** . In short, **well-defined roles, structured communication, and an overseeing orchestrator** create a robust framework where agents collaborate and verify each other's contributions, rather than working in isolation.

Multi-Phase Development Workflow (10+ Steps)

Using the above principles, we can establish a **multi-step workflow (on the order of 10–20 steps)** that an autonomous agentic system should follow to guarantee quality. Below is an example of a comprehensive pipeline incorporating all critical phases from project inception to monitoring, with potential agent roles indicated for each step:

1. **Requirement Analysis & Strategizing:** An agent (or the lead orchestrator) starts by **clarifying the project goals and constraints**. It gathers requirements (user stories, specifications) and determines the success criteria for the project. This may involve querying the user for details or reading a design document. The outcome is a high-level **project strategy**: what needs to be built, key components, and priority order. Having a solid understanding upfront is critical; any ambiguity here can cascade into flaws later. Some implementations use **Multi-View Problem Reading**, having multiple agents independently interpret the requirements and then reconcile any discrepancies . This ensures the project starts on the right foot with a correct, shared understanding of the goals.
2. **Project Planning & Task Decomposition:** Next, a planning agent breaks down the project into a hierarchy of **epics, features, and tasks**. This is akin to project management: determine the major modules or "epics" needed, then split those into smaller **sets of tasks** that can be implemented and tested incrementally. For example, if building a web app, epics might be "Frontend UI", "Backend API", "Database schema", etc., each further divided into tasks (UI component X, API endpoint Y, etc.). The planning agent should output a structured plan (e.g. a list of tasks with descriptions). This plan acts as a roadmap and also as a checklist for the orchestrator to make sure all parts are eventually addressed. If using a hierarchical system, the top-level *Project Manager agent* oversees this and might assign each epic to a specialized *Architect agent* for more detailed design. This explicit planning helps guarantee coverage of all requirements and prevents the agent from jumping straight into coding without a full picture. Each task description also guides the coding agent later on, reducing guesswork during implementation.
3. **Design & Solution Architecture:** For each feature or set of tasks, an **Architect or Designer agent** develops a solution approach before any code is written. This can include creating high-level architecture diagrams, choosing

algorithms or data structures, and even writing pseudocode. Essentially, this is the "thinking" step the user alluded to. The agent should reason about how to implement the requirements: e.g. if a sorting functionality is needed, decide which sorting algorithm to use and why. The design agent produces artifacts like pseudocode, interface specifications, or class outlines. Critically, this design can be **reviewed by a peer agent** (or the planning agent) to ensure it matches the requirements (preventing semantic drift). For example, a Round-Trip Review Protocol could be applied here: have another agent or the same agent in a reflection mode take the proposed design and **reconstruct the original requirements from it**, checking if they align . If the design omits something from the spec or adds extraneous features, the discrepancy is caught **before coding begins** . This step injects a healthy "tension" by not trusting the first design blindly – the system effectively asks, "does this plan solve the intended problem?" and if not, revises the design. By iterating design and review until alignment is high, we lay a solid foundation for implementation.

4. **Implementation (Coding) Stage:** Once a design is approved, a **Coder agent** handles the actual coding. This agent takes one task at a time (according to the plan) and writes the code for it in the required language, following any style guidelines provided. The coder agent should leverage tools as needed – for instance, calling a **terminal tool** to create files or using a compiler/interpreter to sanity-check the code. In advanced setups, the coder could also run basic linters or static analyzers during coding to catch syntax errors or obvious bugs. The key here is that the coding agent isn't operating in a vacuum: it has the design blueprint to follow, and it knows that its output will be tested and reviewed, so it has incentive to write clear, correct code. If multiple coder agents are available, tasks could be implemented in parallel (but on a single machine like an M1 Mac Mini, parallelism might be limited by resources). The orchestrator ensures that as each piece of code is produced, it is stored in the codebase (e.g., a git repository or in-memory file system) ready for testing. **Note:** At this stage, it's vital that the coder agent documents its code appropriately (docstrings, comments) because a later documentation step or maintenance agent might use these. Some frameworks include a separate Documentation agent, but in resource-limited contexts, the coder can at least produce basic documentation alongside implementation.

5. **Unit Testing & Verification:** After coding a component, the pipeline should immediately subject it to **unit tests** written and executed by a specialized **Testing agent** (or a distinct mode of the same agent). Unlike trivial tests that simply mirror the code's expected output, the testing agent's mandate is to **validate the code against the requirements and uncover edge-case bugs**. To ensure these tests are meaningful:

- The testing agent should derive test cases from the **spec and design**, not from the code's internal logic alone. For example, if the component is supposed to handle various input ranges or error conditions (as per requirements), the tests must cover all those scenarios (including boundary values, invalid inputs, concurrency if applicable, etc.).

- Use an *adversarial mindset*: the agent should intentionally try to break the new code. In research, this is called **adversarial testing** – generating inputs that a naive solution might get wrong. One method is **Cross-Test Adversarial Pairing**, where the system creates complementary sets of test cases including both typical cases and tricky edge cases that violate assumptions . The testing agent can also reference common pitfalls or known bugs from similar projects (if such data is in its training or provided via a knowledge base) to craft challenging tests.
- Crucially, the tests must be run in a realistic manner. If possible, the agent should execute the code in an environment similar to production (for example, using the actual runtime via terminal commands on the Mac Mini). This is what the user describes as "real live-fire" testing – not just static analysis or type-checking, but actually running the function/module with various inputs and validating outputs. By running the code, the agent can catch runtime errors, performance issues (like extreme slowdowns), or integration problems (if this code calls other parts of the system).

By making the testing phase robust, we **enforce tension**: if the code is incorrect or incomplete, these tests should fail and flag the issue. The pipeline must then pause and hand control back for fixes (either the coder agent debugs the code, or in some cases the testing agent can suggest fixes too). This aligns with the idea of *test-driven development*: the true completion criterion for the coding step is not "code written" but **"code passes all rigorous tests"** . In fact, agent-based systems have used strict test enforcement to great effect – for example, the SCGAgent framework reports that integrating *iterative code reinforcement with unit-test-backed functionality checks* preserved ~98% of baseline functionality and improved security by ~25% compared to letting an LLM code without tests . In our pipeline, if any unit test fails, the orchestrator should send the task back to the coding agent (or perhaps a debugging agent) to fix the issue, then re-run tests, looping until the unit tests all pass. This closed-loop ensures no step is marked "done" until it truly meets its specification. It prevents the agent from sweeping errors under the rug just to move forward.

6. **Integration Testing (System Testing):** Beyond isolated unit tests, we should also include broader **integration or system-level tests** once a set of components is built. A Testing agent (or a higher-level QA agent) can assemble the newly written modules into the larger application context and simulate real-world use. For instance, if multiple functions form an API, spin up a local instance and send actual requests, or if it's a library, write a small driver program that uses the library like a user would. This **"live-fire" testing** validates that the pieces work together and meet the end-to-end requirements. Integration tests might catch issues like mismatched data formats between components, performance bottlenecks, or state management bugs that unit tests (focused on single functions) wouldn't see. It's important that these tests are not skipped – even if all unit tests passed, integration tests often reveal gaps (e.g., the code works for

each function alone but fails in sequence or at scale). By incorporating a system testing phase (automated by agents), we mimic a staging environment where the product is exercised in full. The agent can use the terminal to run the full program or service and verify outputs, and possibly use monitoring tools to check memory/cpu usage if relevant. This stage adds another layer of quality assurance ensuring the code isn't just a collection of working parts, but a working whole.

7. **Code Review & Refinement:** Even when code passes tests, a **code review** step by a *Reviewer agent* can significantly improve quality and maintainability. This agent's role is to scrutinize the code for best practices, readability, consistency with project style, and any subtle issues that tests might not cover (e.g., security vulnerabilities, scalability concerns, or just poor code clarity). The reviewer agent can be prompted with guidelines like "act as a senior engineer reviewing a pull request – find any problems or suggest improvements." Because it is detached from the coding process, it may catch things the coder agent overlooked. For example, the reviewer might notice duplicated code that could be refactored, or point out that a function's approach is correct but extremely inefficient (which might not fail tests but could be problematic). If issues are found, the system should mark the task for **refinement**: either automatically let the coder agent refactor per the suggestions or loop into a discussion between the coder and reviewer agents to decide on changes (similar to how ChatDev's roles reach consensus through multi-turn dialogue). Many agent-based coding studies have noted that AI-generated code still benefits from this kind of review oversight – in fact, on GitHub, a large share of agent-generated pull requests do get human review before merge; one study of Claude-generated PRs found ~84% were accepted, but around half of them needed minor changes suggested by human reviewers for full compliance with project standards . By incorporating an *automated* review stage, we attempt to simulate that human-in-the-loop sanity check. This helps guarantee that the code isn't just functionally correct, but also clean, well-documented, and aligned with the broader project norms. If the reviewer approves the code with no issues, we can be more confident in its quality.

8. **Documentation & Pull Request Preparation:** Once implementation and verification are complete for a chunk of work, the system should prepare it for integration into the main codebase. This involves two things: **documentation** and **creating a pull request (PR)** (or analogous change submission). A Documentation agent (or the coder agent in documentation mode) can generate any needed documentation: updating README files, writing usage examples, or ensuring inline code comments are sufficient. Simultaneously, the orchestrator or a DevOps agent can create a **PR** that includes the new code changes along with a summary of what was done, referencing the task or issue it addresses. This PR description should be clear and professional, as if a human developer wrote it, because it will serve as a log of changes. In fully autonomous mode, this PR could potentially be auto-merged if tests and reviews all passed. However, many pipelines might still leave the final merge decision to a human operator or at least a final consensus check among agents (see next step). The key point is that by formalizing the code

submission (even in a local git repository on the Mac Mini), we treat the agent's output just like a human contribution – subject to all the normal checks (tests, review, docs) before becoming part of the product. This step ensures traceability and maintainability: future agents (or developers) can read the PR notes to understand why the change was made. It also helps if something goes wrong later; having documentation and PR history aids debugging and accountability, which are facets of quality in the long term.

9. **Consensus & Merge (Approval Phase):** Before deployment or release, it's wise to have a **final approval checkpoint**. In a human setting this is where a tech lead or QA might sign off on the release. In an autonomous context, one can implement a **multi-agent consensus voting** mechanism – essentially, all the specialist agents (or a subset, like the planner, tester, reviewer) give a verdict on whether the code is ready or not. This concept is embodied in frameworks with **Asynchronous Voting Resolution (AVR)** . Each agent can vote "approve" or "needs changes" with some confidence, and only if a weighted consensus threshold is met do we consider the code ready to merge/deploy . If the consensus is not reached, it triggers a re-evaluation loop where the agents revisit outstanding concerns (maybe running additional tests or re-reading requirements) before voting again . This step acts as a safety net against any single agent's blind spot. For example, the code might pass tests and the coder thinks it's done, but perhaps the planning agent or requirements analyst notices a corner of the spec that wasn't implemented. Their "reject" vote would force the pipeline to address that (e.g., implement the missing piece or add a test for it) before final approval. In practice, on a single Mac Mini with a handful of agents, this voting could be as simple as: orchestrator asks each role "do you have any objections?" and if any do, loop back to fix them. But the formal voting concept is how research ensures **no critical perspective is ignored**. Once all agents (or the majority) are satisfied, the orchestrator can consider the work **fully done and merge the PR** into the main branch.

10. **Deployment, Monitoring & Feedback:** With the code merged, the next phase is deployment (if it's a complete application or service) and ongoing monitoring. Deployment might be automatic (e.g., the pipeline could containerize the app and launch it locally or on a test server). Given the Mac Mini constraint, deployment could simply mean running the software locally as a "production simulation". The important part is **Monitoring**: an agent should observe the running system or user feedback channels for any issues that arise post-release. This could involve monitoring logs for errors/exceptions, tracking performance metrics, or scanning user reports. For example, a monitoring agent might periodically run the application's critical functionality and ensure it still responds correctly in a production-like environment (kind of like synthetic testing in production). If any anomaly is detected – say the memory usage spikes indicating a leak, or a new edge-case error appears in logs – the monitoring agent flags it and potentially opens a new task (which goes through the pipeline again to fix the issue). Monitoring closes the loop of the development lifecycle, enabling

continuous improvement. The system effectively doesn't stop at "code merged"; it keeps an eye out for real-world problems and feeds them back as new requirements or tests. This mindset addresses the user's concern about "real live-fire production testing" – by monitoring actual execution in a live setting, the agents can engage in *ongoing testing under real conditions* and catch things that predefined tests might have missed. Additionally, monitoring ensures that our definition of "quality" isn't static; as the software runs over time, any degradation or new bug is caught and handled, maintaining the code's reliability in the long run.

Note: The above workflow is extensive, and in practice one might streamline it depending on the project scope. However, each stage contributes to guaranteeing quality: from double-checking requirements, to designing before coding, to multi-level testing, to review and documentation, and post-deployment checks. The key is that **the agent system is forced to respect each step** – no skipping without careful consideration – and each step has its own success criteria (e.g. design must match requirements, code must pass tests, etc.) that must be met before moving on. By structuring the pipeline in this way, we instill discipline in the AI agents akin to a professional software engineering process.

Robust Testing & Feedback Loops vs. Trivial Tests

Because the user specifically highlighted the issue of agents writing tests "meant to be passed" (i.e. trivial, non-challenging tests), it's worth delving deeper into how to enforce **meaningful testing and iterative feedback** in an autonomous setup. The goal is to create productive *tension* between the implementation and verification steps so that errors surface and get fixed, rather than being glossed over. Several strategies can help achieve this:

- **Independent Test Agent:** Ensure that the agent writing or selecting tests is *not* the same as the agent that wrote the code (or at least is run in a different mode where it doesn't have the code's internal reasoning in mind). This mimics the independence of a QA engineer who approaches the software from a user/spec perspective. An independent test agent will be less likely to "go easy" on the code. In MetaGPT's approach, a dedicated Tester role focuses on validating outputs against requirements , and in practice this agent should be encouraged (via its prompt/SOP) to think of edge cases and adversarial scenarios. For example, if the coder agent produced a function that sorts numbers, the tester agent should try feeding it an already sorted list, a reverse-sorted list, a list with repeated numbers, an empty list, very large list, etc., checking each outcome, rather than just one sorted example.
- **Adversarial Test Generation:** Draw on techniques from research like symmetry-guided or adversarial case generation . The idea is to deliberately stress the code. If the spec or design identifies certain critical properties or symmetry (e.g., if input X and Y should produce the same result, or if a certain

invariance should hold), the test agent can generate cases that *break* those assumptions to see if the code properly handles them . Modern frameworks distribute such adversarial tests *throughout* the development pipeline, not just at the end . For instance, after the design phase, one could already generate some hypothetical test scenarios to validate the design's approach. During coding, after each function, generate targeted tests for that function. After integration, generate another set of more complex tests. This continuous infusion of adversarial evaluation prevents the scenario where the code sails through all stages only to fail on a tricky input post-deployment. It also provides faster feedback to earlier stages. If a design is flawed, an adversarial test might catch it before time is wasted coding it. If the code is flawed, catching it at unit test is cheaper than catching it in system test or production. Essentially, **test early, test often, and test cleverly**.

- **Round-Trip Consistency Checks:** Besides testing functionality, another kind of verification is consistency or **round-trip checks** between adjacent phases. We discussed earlier the Round-Trip Review Protocol (RTRP) concept – for example, verifying that the code indeed implements the design as intended . One practical way to do this is to have an agent read the **code** after it's written and generate a brief summary or **reconstruct the requirements it believes the code is addressing**, then compare that to the original requirements. If the summary misses something or describes a behavior that wasn't intended, that indicates a misalignment. This is a form of "static" verification. It introduces a feedback loop: the code is not only tested dynamically, but also reviewed for conceptual alignment. If misalignment is found, the orchestrator could send the issue back to either redesign or recode as appropriate. This prevents a scenario where the code passes tests (perhaps because the tests were incomplete) but actually doesn't solve the real problem. By having to literally explain what the code does and double-check it matches the spec, the agent system self-audits its fidelity to the task. In our pipeline above, this was touched on in the design stage (checking design vs spec), but it can be applied at multiple levels (design vs spec, code vs design, etc.). These **bidirectional checks** stop semantic drift and ensure each stage's output is anchored to the original goals .
- **Iteration and Backtracking:** A robust pipeline must allow going **backwards** when problems are found. If a test fails in stage 5 or 6, the system should not just log it and move on; it must treat it as a directive to revisit the code (or even the design). This seems obvious, but naive agent systems might not be set up to iterate – they might simply output a test report and end. To guarantee quality, the agents need an automated **debugging loop**. For example, when a unit test fails, the coder agent can be invoked in a "debug" mode to analyze the failing test and patch the code. The system might even spin up a specialized *Debugging agent* that excels at reading error traces or differences between expected and actual outputs to propose a fix. After the fix, tests are re-run, and the loop continues until tests pass. Similarly, if the code reviewer finds a serious design flaw, it might require going all the way back to adjusting the plan or design. This is

essentially implementing a simplified version of how human teams handle feedback: nothing is final until it's correct. Automated pipelines like the **Cross-Verification Collaboration Protocol (CVCP)** implement such loops – they keep iterating design, code, test cycles until either a time/cost budget is hit or the solution meets all criteria . In CVCP, for instance, after each iteration of design -> code -> adversarial test, there's a decision point (voting) that either accepts the solution or sends it back for refinement . Our autonomous pipeline should do the same: treat any unmet quality criteria as a trigger to *repeat or refine the earlier stages* rather than considering them done. This prevents the agent from trying to "skip" genuinely hard work; if it does a sloppy job, the subsequent steps will catch it and force a redo, which creates an incentive (even for a non-sentient system) to just do it right the first time to avoid costly rework.

- **Metrics and Checkpoints:** It may also be useful to establish quantitative **quality metrics** that the agents strive for at each stage. For example, one could enforce code coverage metrics: the test agent might report what percentage of the code was executed by tests, and require (say) >90% coverage before considering the testing adequate. Similarly, one can have performance benchmarks: require that all functions execute within X time or memory. Security linters could ensure no high-risk functions or patterns are present. By making these explicit gates, the agent has concrete targets beyond just "pass a test". In agentic coding research, such evaluation functions are often used to guide the search for better solutions . In practice, your orchestrator can maintain a checklist: e.g. "All unit tests passed (✔), Code coverage ≥ 90% (✔), Lint checks (✔), No high-severity security warnings (✔), Peer review approved (✔)". Only when all checks are green does the pipeline proceed to deployment. If any is red, backtrack and fix it. These automated gates guarantee a baseline of quality.

By combining independent adversarial testing, round-trip consistency checks, and iterative feedback loops, we convert what was once a **rubber-stamp testing phase** into a genuine quality assurance gauntlet for the code. The result is that issues are found and resolved within the agent pipeline itself. Indeed, experiments have shown that introducing such measures can dramatically improve outcomes. For instance, the CVCP framework (which employed symmetry-aware adversarial tests and cross-checks) was able to improve pass rates on hard competitive programming problems by up to **1.8× compared to the simple sequential pipeline** . This underscores that thorough testing and verification at multiple stages are key to **guaranteeing reliable quality** from coding agents.

Project Management & Context Preservation

A critical aspect mentioned is the **hierarchical structure of project management**: handling projects, epics, tasks, etc., in a way that the agent system can manage larger endeavors without losing track of the vision. This goes hand-in-hand with ensuring code quality, because a well-organized project is less likely to produce chaotic or inconsistent code. Here are some best practices on that front:

- **Epics and Milestones:** The orchestrator (or a Project Manager agent) should maintain a high-level view of progress across all epics/features. By structuring work into epics, the system can focus on one major component at a time while keeping others in mind. Epics can be scheduled such that foundational ones are completed first (to avoid constantly revisiting lower layers). When an epic is completed (passes all its tests, reviews, etc.), the orchestrator can mark a milestone. This segmentation helps in two ways: (1) If the project is large, it allows partial integration testing after each epic, ensuring that adding each module doesn't break existing functionality. (2) It provides natural breakpoints to **review architecture** and design consistency across epics. For example, after completing the "Backend API" epic and the "Frontend UI" epic, the system can perform an integration test of those together, and a design consistency check (are data formats consistent? do error messages propagate correctly from backend to UI? etc.). Catching inconsistencies between epics is important; a hierarchical plan where each epic was designed in isolation might have mismatches when they converge. Periodic project-level reviews by an Architect agent can prevent such issues.
- **Persistent Memory & Context:** Unlike a human team that remembers decisions over months, AI agents have limited context windows. To emulate the long-term memory of project context, the system should use tools like a *vector database or knowledge base* to store important information: the overall requirements, decisions made (design choices, trade-offs), and summaries of each component. Then, at the start of any agent's task, relevant context can be retrieved and fed into the prompt. For instance, when a coder agent starts working on epic B, the orchestrator can fetch the design summary of epic A (if relevant) and remind the coder "Note: Epic A uses JSON for data exchange, so Epic B should do the same" – thereby ensuring consistency. This persistent memory could also store known issues or lessons learned as the project progresses. If a bug was found and fixed in one module, the memory can note it, and later when another module is being developed, the agent can be warned if a similar pattern arises. Utilizing such **structured memory** is part of agentic coding best practices , enabling agents to maintain coherence over long, multi-step workflows. It reduces the risk of the agent forgetting constraints or repeating mistakes across tasks.
- **Standardizing Style and Conventions:** Ensuring high code quality at project scale means all agents adhere to the same conventions. This can be enforced by using a **shared coding standard** document that all coding and reviewing agents refer to. It would include things like naming conventions, error handling policies, how to structure unit tests, etc. The Project Manager agent (or orchestrator) can enforce that this document is included in the context of relevant steps (like a style guide prompt for the coder and reviewer). By doing so, even if tasks are done independently, the output remains uniform. In MetaGPT,

embedding human SOPs for each role had a similar effect – it ensured each "team member" followed industry best practices . For our pipeline, an example could be: before coding any file, the coder agent's prompt includes, "Follow the project's coding guidelines: [bullet list of key rules]." The reviewer agent can have the same list to check against. This is a simple measure but it guarantees that code from different parts of the project doesn't conflict or vary wildly in style, which is important for maintainability.

- **Version Control & Issue Tracking:** It's advisable to integrate version control at the core of the pipeline. Every change the agents make can be committed to a git repository (even a local one) with appropriate messages. This not only provides a safety net (the ability to roll back if something goes wrong) but also instills discipline in changes – each commit should relate to a specific task or fix. An **issue tracker** (which could be as simple as a list of to-do items the orchestrator maintains, or a full-featured system like GitHub Issues if using online) helps keep track of tasks and their status (open, in progress, done). The orchestrator can update these statuses as the pipeline runs. This mimics project management tools used in real software projects (Jira, Trello, etc.), giving transparency to what the autonomous agents have completed and what remains. It also enhances reliability: if the system crashes or needs to be stopped and restarted (maybe due to compute limits), the issue tracker + git history allow it to resume work without forgetting what was done. Essentially, the agents are continuously documenting their project management state, which is crucial for a multi-step, long-running process.
- **Human Oversight at Milestones:** While the aim is autonomy, *guaranteeing* quality might still benefit from occasional human checks, especially at major milestones. The user did not emphasize human-in-the-loop, but it's worth noting as a best practice: one can insert optional pause points after each epic or before deployment where a human can review summaries or diff outputs. Even if not utilized, designing the pipeline to allow this (e.g., by outputting a concise report at each milestone) means the process is transparent and auditable. This can catch any systematic deviations early. For example, if after the first epic, a quick glance by a human or an external audit tool shows the code is too slow or using deprecated functions, the strategy can be adjusted. The need for this will diminish as the agents become more reliable, but in the current state of AI, it's a prudent fallback to ensure nothing critical slips by. In fact, empirical studies of agent-generated code in real OSS projects found that **human oversight still adds value** – many AI contributions were merged only after humans refined them for edge cases or style consistency . Our pipeline, by mirroring a rigorous human process, tries to minimize the need for human fixes, but keeping the door open for oversight is wise if absolute guarantee is needed.

By managing the project with hierarchical decomposition, memory of decisions, and consistent standards, the agent orchestration can handle complex projects systematically. Each sub-task is completed with context and quality in mind, and the bigger picture is never lost. This **project management discipline** is as critical as the technical testing for achieving high-quality results – many issues in software arise not from coding errors per se, but from miscommunications, forgotten requirements, or integration mismatches in larger systems. A well-orchestrated agent system, therefore, treats project management artifacts (plans, design docs, style guides) as first-class citizens that guide the coding, rather than jumping straight into writing code and tests blindly.

Resource Efficiency & Feasibility on Local Hardware

The user specifically mentioned the solution must be workable on an **M1 Mac Mini** without over-stressing computational limits. This constraint affects how we design the agent orchestration in practice. While the principles above remain the same, we should consider optimizations to ensure the process runs within reasonable time and resource usage:

- **Model Selection:** Using very large LLMs for every agent and every step might be impractical on local hardware or even via API (due to cost). One approach is to **mix model sizes** according to task criticality. For instance, planning and design might benefit from a powerful model (for complex reasoning), but routine code generation or refactoring could possibly be done by a smaller, fine-tuned code model that can run locally. If the user has access to OpenAI Codex and Anthropic Claude via APIs, they might already be leveraging cloud compute for the heavy tasks. In that case, the main limitation is cost and latency rather than hardware. But if purely local, one could consider running a smaller model (like Code Llama or another local LLM) for some roles. In any case, orchestrating multiple steps serially is inherently slower than doing one big generation, so optimizing each call is key.
- **Parallelism vs Sequential:** On an 8-core Mac Mini, some parallelism is possible. For example, after a coding step, the system could potentially run unit tests and static analysis in parallel (since running tests via the Python/terminal tool can utilize CPU separate from the LLM). The design of the pipeline could allow certain independent checks to happen concurrently to save time. However, coordinating multiple LLM agents truly in parallel is tricky if using APIs (since you can call them concurrently but have to manage the threads). More straightforward is to minimize *unnecessary* steps. The **Puppeteer-style dynamic orchestrator** we mentioned can come into play here: if an agent determines a particular check doesn't apply, it can skip it to conserve tokens and time . For example, if a change is very small (like fixing a typo), the orchestrator might decide to skip a full redesign phase and just run tests to confirm the fix, thereby speeding up the cycle. Similarly, if a piece of code has 100% coverage from previous tests, the agent might skip writing new tests unless the code changed substantially. Careful caching of results can help – e.g., remember that "function X was already tested thoroughly, so we don't need to retest it unless it was modified". These are more advanced optimizations, but important for long-running processes on limited

hardware.

- **Ephemeral Agents & Resource Reuse:** The Medium article suggested using ephemeral, on-demand agents that spin up only when needed . On a Mac Mini, this concept translates to not holding large models in memory when not needed. If using API calls, this is naturally the case (no persistent memory cost aside from context windows). If using local models, one could load a model, perform a task, then unload it or swap it out for another if memory is a concern. The M1 has unified memory, so running multiple models simultaneously might be limited. It may be better to sequence agent calls (which is fine given the pipeline structure) rather than truly parallel multi-model usage on local. Also, using techniques like prompt-reduction and keeping contexts lean will help avoid hitting context length limits and incurring extra compute. In other words, feed each agent just enough information to do its job (which is where the structured memory retrieval helps) rather than dumping the entire project state on every call.
- **Tool Integration to reduce LLM work:** Wherever possible, use *external tools* to do heavy lifting instead of making the LLM figure it out from scratch. For example, running the code's test suite is something that can be done by actually executing the code – the LLM doesn't need to simulate the code's output mentally (which would be inefficient and error-prone). The user indicated they are using *terminal commands*, which is great – it means the pipeline can compile/run code, run test scripts, etc., directly on the machine. Similarly, static code analysis can be done with linters or analyzers (like flake8 or mypy for Python, etc.) instead of asking the LLM to spot all style issues. The LLM agent can just interpret the tool outputs and make decisions. This aligns with best practices in agentic coding where **dynamic tool usage** is encouraged to improve efficiency and accuracy . The LLM focuses on decision-making and creativity (like writing code or interpreting results), while the heavy computation (running the code, calculating coverage, etc.) is offloaded to traditional software. By doing so, we reduce the token usage and the cognitive load on the agent, making the process faster and more reliable.
- **Cost/Benefit of Steps:** To keep things efficient, one could **configure the level of rigor** based on the criticality of the task. For core or high-risk components, you run the full gauntlet (extensive design, multiple review iterations, large test suites). For minor or low-risk components, you might accept a lighter process (maybe fewer tests or a quicker review). This is similar to how humans apply more scrutiny to complex code than to trivial boilerplate. The orchestrator can be given rules or learn heuristics about this. For example, if a task is labeled "refactoring" or "documentation update", perhaps it doesn't need integration tests, just ensure nothing broke. On the other hand, if a task is "implement new payment processing logic", that likely needs maximum scrutiny. This dynamic allocation of effort helps in not overloading the system for every single change. It maintains quality where it matters most, and conserves resources on less critical changes.

In summary, an agentic coding pipeline **can be run on a local Mac Mini** by carefully managing the models and computation. The focus should be on smart orchestration (do only what's necessary, when necessary) and leveraging the machine's capabilities (running code, parallelizing independent tasks) to complement the LLM's reasoning. By trimming inefficiencies, we ensure that the pursuit of quality does not grind the system to a halt or exhaust the hardware. You get a reliable outcome without needing a supercomputer – just intelligent coordination.

Conclusion & Outlook

Building a system for autonomous coding and orchestration that guarantees high-quality code is an ambitious but increasingly achievable goal. By instituting a **multi-step, multi-agent workflow** – from initial strategy through planning, design, coding, testing, review, and deployment – we compel the AI to engage in a thorough software engineering process rather than shortcutting its way to a quick answer. Each stage in the pipeline serves as a checkpoint to maintain alignment with requirements and catch errors early. The use of specialized agents (or specialized modes of a single agent) for each role introduces natural checks and balances, as one agent's output is vetted by another's expertise . This agent team approach, inspired by human teams, has already shown tangible improvements in research settings (e.g. higher correctness and acceptance rates of generated code) and in practical use (autonomous PRs getting accepted with minimal edits) .

Crucially, **robust testing and validation** is the linchpin of quality assurance. By moving beyond trivial, self-fulfilling tests to real "battle-tested" scenarios – including adversarial cases and integration tests – the agents are forced to truly prove their code works as intended. Any tendency to write code that "only passes my simple test" is countered by an independent test agent that **breaks** that code if it's fragile. The iterative loops ensure the system actually fixes issues rather than papering them over. In effect, the AI agents are guided to act less like passive code generators and more like proactive engineers who check and double-check their work.

Furthermore, introducing hierarchical planning and memory into the process means the **context and big picture are preserved** throughout development. The project's goals inform every decision, reducing the chance of divergence or regression. Each component is built with knowledge of the whole, and each improvement is recorded and monitored. By structuring tasks into manageable units (projects → epics → tasks) and keeping a vigilant eye (via monitors and feedback), the pipeline can scale to complex projects systematically.

It's worth acknowledging that even with all these measures, fully "guaranteed" quality is an ideal – in practice one should always have a contingency for unforeseen issues. Nevertheless, the combination of techniques described –

borrowed from cutting-edge frameworks like MetaGPT, ChatDev, and CVCP – pushes significantly closer to that ideal by **minimizing blind spots** in the development cycle. Indeed, recent studies have noted that these agentic coding approaches can improve performance on challenging programming tasks by substantial margins compared to naive one-shot coding , and industrial reports show a large proportion of AI-generated code can meet acceptance standards when proper process is enforced .

Finally, as a forward-looking note: this field is evolving rapidly. We can expect even more sophisticated methods for agent collaboration (such as learned dynamic orchestration, better self-evaluation metrics, and maybe lightweight formal verification integrated into the loop). Concepts like **self-improving agents** that learn from each coding attempt to do better next time are emerging . Incorporating such advancements in the future could further increase reliability – for example, an agent could learn which types of tests caught bugs historically and automatically focus on those patterns in new projects. For now, the described approach provides a strong foundation: a disciplined, multi-agent workflow that, even when running on a modest machine, compels AI coders to produce work that is *thoroughly vetted, aligned with requirements, and maintainable*. By treating each step with the seriousness a human expert team would, we significantly close the quality gap between autonomous code generation and traditional development.

With this setup in place, one can have greater confidence that the code coming out of the pipeline isn't just *correct by construction*, but also robust in the face of real-world usage – fulfilling the promise of autonomous agentic coding with high quality.

Sources: Recent research and industry insights were referenced to support this approach, including the definition and principles of agentic coding , case studies of multi-agent development frameworks , analyses of pitfalls in naive pipelines , and advanced techniques like cross-verification, adversarial testing, and consensus protocols for reliability . These demonstrate the efficacy of structured, rigorous agent orchestration in improving software outcomes. Each element of the proposed workflow is grounded in practices shown to enhance either the correctness, security, or maintainability of AI-generated code , aligning with the user's goal of **"guaranteed reliable quality"** from autonomous coding agents.

Got it—here's a tight, practical pass on the **cutting edge you can actually use** for autonomous/agentic coding quality, with concrete techniques, tools, and repo-ready wiring. I'll focus on the bits that raise quality *during* execution, preserve *coherence* across steps, and produce tests that *fight back* instead of rubber-stamping.

---

1) Guardrails that work in practice (today)

A. Chain-of-Verification, not just Chain-of-Thought

Make agents **draft → plan checks → answer checks independently → revise**. This reduces "confidently wrong" outputs and stops the rush to merge. Treat it as a required sub-routine in Plan, Implement, and Review. Empirical work shows CoVe reduces hallucinations across tasks; translate that into code by turning the "checks" into assertions, quick scripts, or targeted unit cases.

Drop-in: add a "Verification Plan" block to every agent step (e.g., "Which invariants/edge cases could falsify this change?"). In CI, reject a step if it lacks verification Q&A artifacts.

B. Symmetry-guided adversarial testing (SGAT) + round-trip reviews

Use *symmetries* and *invariants* of the spec to auto-mint **counterexamples** and "mirror inputs" that should yield equal or transformable outputs (e.g., encode/decode, sort/unsort, add/remove noop). Combine with **Round-Trip Review Protocol (RTRP)**: "From the code, reconstruct the requirement—does it match?" Multi-agent protocols that mix these (CVCP) improved pass rates on hard programming tasks by forcing tension between steps. Wire this into your test writer so it never produces only "happy path".

Drop-in: for each task, have a small library of symmetry macros (invertible transform, permutation, idempotence, commutativity where applicable). The test-agent must instantiate at least N symmetry cases before PR.

C. Mutation-guided test generation (but cheap)

Don't just measure coverage—measure whether tests **kill mutants**. The 2025 wave is using LLMs to *prioritize* a **few, high-value mutants** and then generate targeted tests, giving better fault detection without huge compute. Use it to "stress" agent-written code *and* the agent-written tests.

Drop-in on M1:

- Python: mutmut or cosmic-ray with a *prioritizer* script (LLM suggests likely fault spots based on diff + spec); generate tests only for those.

- JS/TS: Stryker.
- Java: PIT.

Run a small mutant budget per PR (e.g., ≤30) and gate on “mutation score didn't regress.”

D. Property-based tests (PBT) as a first-class artifact

Have the test-agent extract **properties/invariants** from spec/ADRs and write PBT with Hypothesis (Python) or fast-check (TS). Recent studies show LLMs can generate workable PBTs and that PBT improves edge-case discovery versus example-only tests. Use PBT to reflect the *intent* of the component, not its current behavior.

Drop-in: require every non-trivial module to include at least one PBT that encodes an invariant; forbid merging if only example tests exist.

E. Multi-agent roles with hard gates (MetaGPT/ChatDev pattern)

Use a **planner → coder → tester → reviewer** pipeline with explicit SOPs and hard pass/fail gates between roles (no skipping). This structure (validated in academic frameworks) raises completeness and consistency by forcing cross-checks.

Drop-in: make “Cited Plan” mandatory (planner must cite ADRs/tests/specs it retrieved); make “Reviewer Objections = 0” mandatory (review agent lists defects; if any remain, loop back).

---

2) Context coherence across a large repo

A. Hybrid semantic retrieval, symbol-aware

Use **AST-level chunking** for code/docstrings/tests, **heading-based** for docs/ADRs, and a **hybrid** retriever (BM25 + vectors) with a **cross-encoder reranker**. Group results by **intent** (ADRs, closest symbols, related tests, past incidents). This prevents long-context soup and keeps the agent tied to governing decisions. (You already asked about this; this is the “ship it” version.)

Why it matters: it's the difference between tests aligned to *intent* versus tests that just make the current code pass. (Benchmarks like SWE-bench highlight how lack of real repo context tanks success.)

Stack on M1: FAISS + SQLite/Tantivy, any modern code+text embedding, small cross-encoder for rerank. (Local or hosted; keep k small, cache aggressively.)

B. Decision lineage as a dependency

Treat **ADRs/specs/incident postmortems** as dependencies: plan/coder/tester must cite them; PR template includes "Related ADRs/PRs/Incidents." In CI, run a "semantic impact set" to find siblings that *should* co-change (and fail if you didn't touch them). This is how you enforce *project-level* coherence.

---

3) "Live-fire" verification without blowing compute

- **Run code and tests for real.** Use the terminal tool to execute modules in a **prod-like script** (not just import-time). Combine with low-cost **PBT fuzz** (Hypothesis minimal draws) and **small, prioritized mutants** per PR.
- **Static + semantic linters as gates:** Semgrep/Bandit (security), mypy/ruff/eslint (correctness/clarity). Reviewer agent must reconcile linter findings or explain why they're non-issues.
- **Performance/complexity spot-checks:** For hot paths, require micro-bench stubs (pytest-benchmark) and assert "no worse than X% over baseline".

---

4) Benchmarks & reality check (so you know what "good" looks like)

- **SWE-bench / SWE-bench-Live** are the de-facto reality checks for repo-scale issue solving. Use them to validate your orchestration settings (agent count, gating strength, retrieval config). Don't expect miracles; even top agents struggle unless retrieval and test quality are strong.
- "Autonomous engineer" products (Devin/OpenHands) are evolving, but reported gains come from **planning+testing+PR loops**, not raw generation. Use their patterns (ticket→plan→test→PR) as structure, not as a black box.

---

5) Minimal, implement-today wiring (M1-friendly)

CI/CD quality gates (GitHub Actions example)

1. **Index (diff-only)** → hybrid search service refresh.

2. **Unit+PBT** (pytest + Hypothesis) → must pass.
3. **Mutation budget** (e.g., 20 targeted mutants via mutmut) → mutation score ≥ team threshold; no regression.
4. **Symmetry pack** (your SGAT library) → N symmetry cases must pass.
5. **Linters/security** (ruff, mypy, semgrep/bandit) → zero high severity.
6. **Reviewer agent** → no unresolved objections; PR must include **Cited Plan** and **Related ADRs**.

Agent SOP snippets (what you put in each role's prompt)

- **Planner:** "Produce Cited Plan: user story, acceptance criteria, impacted symbols, *why now*, ADR/test links."
- **Coder:** "Conform to spec; seed docstrings from acceptance criteria; emit a verify.py entry that exercises primary path."
- **Tester:** "Produce (a) example tests from acceptance criteria, (b) one PBT capturing an invariant, (c) SGAT cases (list symmetry used), (d) targeted mutants killed list."
- **Reviewer:** "Run static tools; check for missed invariants; compare code→spec via round-trip summary; enumerate objections."

---

6) Tools & repos (cutting edge, usable)

- **Multi-agent scaffolds:**
- **MetaGPT** (role SOPs baked in) — adopt the role prompts and gating patterns even if you don't use the framework.
- **ChatDev** (design→code→test with communication discipline) — copy the phase boundaries and "communicative de-hallucination" checks.
- **OpenHands** (ex-OpenDevin) — OSS agentic dev platform with shell/editor/browser control; use its task loop but upgrade its QA gates with the items above.
- **Testing accelerants:**
- **Mutation testing:** mutmut, cosmic-ray (Py); Stryker (JS/TS); PIT (Java). Prioritize mutants (LLM suggests where to poke) following Meta's ACH ideas.
- **PBT:** Hypothesis (Py), fast-check (TS). New work shows LLM-assisted PBT is feasible; keep generators small to fit M1.
- **Verification patterns:**
- **CoVe** (verification Q&A) — turn questions into assertions/tests.
- **SGAT + RTRP + voting** — implement the CVCP trio at *small scale*: symmetry tests, round-trip summaries, then a simple "approve/needs-changes" vote from Tester+Reviewer.
- **Benchmarks & trackers:**
- **SWE-bench / Live** to tune your pipeline knobs; don't chase leaderboard SOTA—chase your repo's "defect escape rate".

---

7) How this fixes your exact pain points

- **"Agents write tests meant to pass."**

SGAT + mutation budget + PBT means tests must *break* shallow implementations. Mutation score and symmetry passes become merge gates.

- **"No tension with prior steps; skipping stages."**

Role-gated pipeline (MetaGPT/ChatDev style) where each step must produce **artifacts** (Cited Plan, invariants, objections) that later steps consume; CI fails if any artifact is missing.

- **"Coherence across project hierarchy."**

Hybrid retrieval keyed by ADRs/specs/incidents; "semantic impact set" blocks PRs that ignore sibling symbols/functions that should co-change. Benchmarks reinforce that retrieval quality gates success.

- **"Live-fire > compile-only."**

Require a verify.py/verify.sh per task that spins the module in a realistic way, then run it in CI alongside PBT and mutants.

---

8) Evaluation that reflects true quality

Track these per PR and per epic:

- **Mutation score** (small budget) — gate.
- **PBT pass & shrinkage** (Hypothesis minimized a counterexample? then you found a real edge).
- **Symmetry pack pass rate** — gate.
- **Reviewer objections = 0** after loop — gate.
- **Defect escape rate** post-merge (integration run + any incident backfills a failing test).
- **Retrieval health** (nDCG@k on a tiny gold set of "must-find" ADRs/tests for canonical queries).

TL;DR "gold path" you can wire up this week (on your M1)

1. Add **hybrid search** (diff-indexed; AST+docs); enforce "Cited Plan."
2. Require **PBT + SGAT cases** for every non-trivial change.
3. Run a **tiny, prioritized mutation budget** per PR; gate on score.
4. Use **CoVe** in Plan/Implement/Review prompts; convert its questions into tests.
5. Keep **role gates** (planner→coder→tester→reviewer) and block merges on missing artifacts/objections.

If you want, I'll turn this into a **concrete check suite** (GitHub Actions + Makefile targets + a tiny SGAT helper + Hypothesis scaffolds) that drops straight into your repo and your autopilot.

Here's a concrete blueprint for an **agentic autopilot** that can plan, build, test, review, and continuously refine an entire program—editing its own roadmap while staying laser-aligned to the original purpose, and doing it on an M1 Mac mini without melting your fans.

---

0) First principles (guardrails that make this work)

- **Goal-locked, plan-flexible.** The *purpose* is fixed; the *roadmap* is mutable. Encode the purpose as invariants, acceptance criteria, and non-functional constraints (NFRs). Let the roadmap be a living, ranked backlog the autopilot can rewrite—but every change must cite how it improves purpose/NFR satisfaction.
- **Role separation + hard gates.** Use small, specialized agents (Planner, Coder, Tester, Reviewer, Roadmap-Editor) with pass/fail gates. No role can "skip ahead" without producing required artifacts (Cited Plan, tests, review notes, etc.).
- **Evidence ledger.** Every decision produces evidence (citations, diffs, test results). Merges only happen with green gates and linked evidence. If it's not in the ledger, it didn't happen.
- **Live-fire over compile-only.** Always run the program (or a realistic harness), not just lints. Add property-based tests + a tiny mutation budget per PR.
- **Tight compute loop.** Do the minimum necessary work each iteration: diff-based indexing, small candidate sets, cached retrieval, small batches.

---

1) System architecture

Control plane (one small orchestrator process):

- **Orchestrator** (state machine) runs the macro-loop and enforces gates.
- **Task Router** picks the next work item (from roadmap/backlog).
- **Evidence Store** (SQLite) persists artifacts: plans, diffs, test logs, votes.
- **KB / Retrieval**: FAISS (vectors) + SQLite/Tantivy (lexical), populated by a **Diff Indexer**.

Worker agents (can be the same model with different SOPs):

- **Planner** → decomposes goal/epic into tasks; emits *Cited Plan*.
- **Coder** → implements changes; emits code + verify.sh/py.
- **Tester** → writes example tests + property-based tests + symmetry

cases; runs tests; reports coverage & mutation score.

- **Reviewer** → static/security review, round-trip requirement check, objections list.
- **Roadmap-Editor** → proposes roadmap edits (reprioritize, split/merge tasks) with proof (metrics delta, evidence).

Tools (on M1):

- Python runtime, pytest, **Hypothesis** (PBT), mutmut (or cosmic-ray) with a **small mutant budget** (e.g., 20/PR).
- Linters: ruff, mypy (Py) or ESLint/TS; security: bandit/Semgrep.
- Git (local), Makefile, GitHub Actions (optional), shell access.

---

2) Data & memory model

ProjectPurpose.yml (the North Star):

```
purpose: "High-quality X that does Y for Z users"
acceptance_criteria:
  - ...
non_functional_requirements:
  - reliability >= 99.9% simulated
  - latency_p95 <= 120ms on test workload
  - maintainability: docstrings for public APIs, cyclomatic complexity <= N
guardrails:
  - prohibited_patterns: [...]
  - required_patterns: [...]
```

Roadmap.json (editable by the autopilot):

```
{
  "epics": [
    {"id":"E1","name":"Core API","prio":1,"status":"in_progress","value":0.35},
    ...
  ],
  "tasks": [
    {"id":"T14","epic":"E1","name":"Add exponential backoff","status":"todo",
     "deps":["T09"],"risk":"med","value":0.12,"evidence":[]}
  ]
}
```

Evidence ledger (SQLite tables): decisions, artifacts, test_runs, reviews, votes, mutations, all keyed by git SHA and task id.

KB/Index:

- **Chunkers:** AST-level for code; heading+sentences for docs/ADRs; per-test for /tests; per-comment for PR/issue text.
- **Metadata:** repo, path, commit, symbol, kind, headings, ADR ids, issue/pr ids, owner, timestamps.
- **Hybrid search:** BM25 ∪ FAISS → cross-encoder rerank → grouped by intent (ADRs, closest symbols, tests, incidents).

---

3) Macro loop (the autopilot state machine)

```
LOOP
  Sense:
    - Pull repo diff, update KB (changed files only)
    - Compute health snapshot: test pass %, coverage, mutation score, open objections
  Decide:
    - Select next task (value/risk/blocked?); or let Roadmap-Editor propose reordering
    - If roadmap change proposed, require proof (metrics simulation or constraints improvement) and pass a vote
  Plan (Planner):
    - Produce Cited Plan:
        • goal + acceptance criteria
        • impacted symbols/modules
        • related ADRs/specs/tests (top-k citations)
        • plan-of-attack + verification plan (invariants, symmetry cases, PBT outline)
    - Gate: must cite ADR + at least 1 negative test/incident + target invariants
  Act (Coder):
    - Implement; create or update verify.sh/py; write docstrings
    - Small commits with meaningful messages
  Check (Tester):
    - Generate example tests from acceptance criteria
    - Write 1+ property-based test (PBT) for a named invariant
    - Instantiate symmetry-guided cases (SGAT) for that component
    - Run unit+integration; compute coverage
    - Run a small mutation budget; compute mutation score
    - Gate: tests green AND coverage threshold AND mutation score >= floor
  Review (Reviewer):
    - Run static/security tools; summarize issues
    - Round-trip review: "From code, reconstruct requirement; compare"
    - Gate: zero unresolved objections
  PR & Merge:
    - Create PR with template (Cited Plan, evidence table, risks, rollbacks)
    - If all gates green, merge to main; else loop back to previous phase
  Learn:
    - Update Roadmap.task value/risk using evidence (e.g., defects found)
    - Update heuristics (where mutants died → future testing focus)
END
```

Fail-fast gates (non-negotiable):

- Missing *Cited Plan* with ADR/test citations → block.
- No PBT or no symmetry cases → block.
- Mutation score regression vs baseline → block.
- Reviewer objections outstanding → block.

---

4) How the roadmap can self-edit without drifting off purpose

Roadmap-Editor agent policy:

- May: split a task, merge duplicates, reprioritize, add discovery tasks.
- Must: justify with **evidence**: improved value/effort ratio, risk reduction, or constraint satisfaction (e.g., shaving 40ms p95).
- Must not: alter **purpose** or acceptance criteria. Any suggestion impacting purpose must be rejected by the orchestrator.

Change process:

1. Editor proposes a patch to Roadmap.json with deltas.
2. **Impact preview:** run “semantic impact set” (what else co-changes?), recalc dependency risk.
3. **Vote:** Planner + Reviewer + Orchestrator (simple majority) accept/reject.
4. On accept: commit Roadmap patch; post rationale in ledger.

This gives you *adaptive planning* without goal drift.

---

5) Quality: the four levers you wire in from day one

1. **Cited Plan** → forces intent alignment (Planner must pull ADRs/specs/tests/incidents).
2. **PBT + SGAT** → forces tests that fight back (Tester must encode invariants and symmetry).
3. **Mutation budget** → forces fault-finding beyond coverage.
4. **Round-trip review + security/static** → forces conceptual and non-functional correctness.

All four are cheap enough for an M1 when budgeted (e.g., ≤20 mutants; Hypothesis with small examples; cached retrieval).

---

6) Minimal repo scaffold

```
.autopilot/
  orchestrator.py          # state machine
  sop/                     # role SOPs (prompts with checklists)
    planner.md
    coder.md
    tester.md
    reviewer.md
    roadmap_editor.md
  gates.yml                # thresholds (coverage, mutation floor, lint severity)
  templates/
    pr.md
    cited_plan.md
    review_checklist.md
  tools/
    symmetry.py            # SGAT helpers (inversion, idempotence, etc.)
    metrics.py             # read coverage.xml, junit.xml, mutation.json
    kb_indexer.py          # AST/doc chunker, FAISS/SQLite loaders
  kb/
    faiss.index
    metadata.sqlite
  state/
    ledger.sqlite
ProjectPurpose.yml
Roadmap.json
Makefile
.github/workflows/ci.yml
```

---

7) SOP excerpts (copy/paste and tweak)

Planner SOP (cited plan)

- Inputs: ProjectPurpose.yml, Roadmap task, KB search results.
- Output must contain:
- Acceptance criteria restated

- Impacted symbols/modules (paths, lines)
- **Citations:** ≥1 ADR/spec, ≥1 existing test or incident
- Verification plan: list invariants; symmetry cases; PBT outline
- Exit: fail if any required citation or invariant is missing

Tester SOP (tests that fight back)

- Write example tests from acceptance criteria.
- Write ≥1 Hypothesis test encoding a named invariant.
- Instantiate symmetry cases from tools/symmetry.py (e.g., encode/decode; noop; commutativity/idempotence where applicable).
- Run tests + coverage; run mutation budget (≤20); produce test_report.json with coverage%, mutation score, failing cases (if any).
- Exit: fail if mutation score < floor or no PBT/symmetry present.

Reviewer SOP (round-trip + security)

- Run ruff/mypy/bandit (or ESLint/Semgrep). Flag issues.
- Round-trip: summarize what the code **does**; diff against acceptance criteria; list mismatches.
- Approve only if zero unresolved issues and round-trip aligns.

---

8) CI wiring (lean but strict)

Makefile

```
index:
\tpython .autopilot/tools/kb_indexer.py --diff
test:
\tpytest -q --maxfail=1 --disable-warnings
pbt:
\tpytest -q -k "property_"   # tag PBTs
mut:
\tmutmut run --paths-to-mutate src/ --tests-dir tests/ --use-coverage
lint:
\truff check src tests && mypy src && bandit -r src
verify:
\tbash ./verify.sh || python verify.py
gates:
\tpython .autopilot/tools/metrics.py --enforce .autopilot/gates.yml
```

.github/workflows/ci.yml (sketch)

- Step 1: make index
- Step 2: make test pbt verify

- Step 3: make mut (cap runtime via timeout)
- Step 4: make lint
- Step 5: make gates (fail on thresholds / missing artifacts)
- Step 6: Autopilot posts review (or blocks merge)

---

9) Orchestrator skeleton (pseudo-Python)

```
while True:
   kb.update_diff()
   health = snapshot_health()
   task = select_task(roadmap, health)  # or let editor propose patch + vote

   plan = Planner.cited_plan(task, kb, purpose)
   gate(plan.ok, "Missing citations or invariants")

   diff = Coder×implement(plan)
   run("git add -A && git commit -m 'WIP: {}'".format(task.id))

   tr = Tester.run_all(plan, diff)  # unit+integration+PBT+SGAT+mutation
   gate(tr.green and tr.mutation >= floor and tr.pbt and tr.sgat, "QA failed")

   rv = Reviewer.review(diff, plan, tr)
   gate(rv.ok and not rv.objections, "Review failed")

   pr = create_pr(plan, tr, rv)
   if all_gates_green(tr, rv):
      merge(pr)
      update_roadmap_success(task, tr)
   else:
      log_and_refine(task, tr, rv)
```

---

10) How it avoids your current pitfalls

- **Superficial tests:** PBT + symmetry + mutation **force** robust tests; CI blocks if absent or weak.
- **Skipping steps:** Role gates with artifact checks prevent “fast-forward”; missing Cited Plan or objections ⇒ loop back.
- **No coherence:** Hybrid retrieval (symbol-aware) and **semantic impact sets** demand that related modules/tests change together; planner must cite ADRs/specs.

- **No live-fire:** Every task must ship a verify.sh/py that runs a realistic scenario; CI runs it on every PR.

---

11) Day-by-day bring-up (1–3 days to MVP)

Day 1

- Add ProjectPurpose.yml, initial Roadmap.json.
- Drop .autopilot/ scaffold; implement KB indexer (diff-only), FAISS/SQLite.
- Write SOPs and gates; wire Makefile + CI minimal ("test + lint + verify").

Day 2

- Implement Planner, Coder, Tester, Reviewer wrappers (simple prompts).
- Add symmetry.py helpers; tag at least one PBT in your codebase.
- Add mutmut with a 10–20 mutant budget targeting touched files.

Day 3

- Enforce PR template & gates. Turn on block-on-red.
- Add Roadmap-Editor with vote and evidence requirements.
- Start collecting metrics (coverage, mutation score, objection count, defect escape).

You'll feel the upgrade immediately: agents will slow down just enough to be *correct*, and PRs will come with real evidence.

---

12) Metrics that actually matter

- **Per-PR:** coverage %, mutation score, #symmetry tests, #PBTs, #review objections (should trend to 0), cycle time.
- **Per-epic:** defect escape rate (issues after merge), p95 latency under verify workload, coherent co-changes (ratio of impacted siblings touched vs expected).
- **System:** retrieval nDCG@k on a tiny gold set of "must-find" ADRs/tests; build flake rate; CI wall time.

Final note

You don't need a massive model zoo or swarm to get "elegant, efficient, high-quality" autonomy. You need: **goal-locked planning, role gates, tests that fight**

back, a tiny mutation budget, and a ledger of evidence. Start with the scaffold above, keep the budgets small for your M1, and let the autopilot *earn* every merge.