GRANTRADAR

Technical Build Plan

3-Week MVP Implementation

Version 1.0 | January 2026

Build Overview

Timeline: 3 weeks (21 days) to functional MVP with paying customers

Team: 1 full-stack engineer (you) + Claude Code Agent SDK as development accelerator

Success Criteria: 10 paying beta users at $200/month, <5 minute alert latency, 85%+ match accuracy

Core Philosophy: Leverage agentic workflows to write 70% of boilerplate code. Use Claude Code Agent SDK to generate database schemas, API endpoints, agent workers, and infrastructure config. Focus human effort on business logic and system design.

Agentic Development Strategy

Key Principle: AI agents write implementation code, humans design architecture and provide specifications. This 10x's development speed for infrastructure-heavy projects.

Claude Code Agent SDK Setup

Install and configure Claude Code Agent (available via Anthropic):

- Install: npm install -g @anthropic-ai/code-agent
- Configure: Export ANTHROPIC_API_KEY, set project context
- Create .claudecontext file with tech stack (Python, FastAPI, React, PostgreSQL, Redis, Celery)
- Use: claude-agent <task> - Agent writes code directly to files, runs tests, iterates on errors

What Claude Builds vs What You Design

Human Designs (High-Level)	Claude Implements (Code)
Database schema design (tables, relationships)	SQLAlchemy models, migrations, indexes
Event-driven architecture (streams, queues)	Redis pub/sub, Celery workers, event handlers
API endpoints specification	FastAPI routes, Pydantic models, auth middleware
Scraping strategy (what to scrape, how often)	Playwright scripts, parsing logic, error handling
Matching algorithm design	Vector search queries, LLM prompts, scoring logic
UI component requirements	React components, Tailwind styling, state management

Time Savings: Implementing infrastructure code manually = 12-15 days. With Claude Code Agent = 3-4 days. You save 8-11 days to focus on business logic, testing, and iteration.

Agentic Workflow Pattern

For each component, follow this 4-step workflow:

1. **Specify:** Write detailed spec in natural language (see examples below)
2. **Generate:** Run claude-agent with spec, it writes code and tests
3. **Review:** Check generated code for correctness, suggest improvements
4. **Iterate:** If issues, provide feedback → agent fixes → repeat until working

Example Iteration:

You: 'Build PostgreSQL schema with grants, users, lab_profiles, matches, alerts_sent tables. Use pgvector for embeddings.'
Claude: [Generates models.py with SQLAlchemy models]
You: 'Add indexes on grants.posted_at and grants.deadline for faster queries'
Claude: [Updates models.py with Index declarations]

Week 1: Real-Time Infrastructure

Goal: Build event-driven backend that can detect new grants within 5 minutes and store them in database

Day 1-2: Core Infrastructure Setup

Database Schema (PostgreSQL) - Using Claude Code Agent

Step 1: Write Specification

Create file: *specs/database_schema.md*

```markdown
# Database Schema Specification
Build PostgreSQL schema with SQLAlchemy models and Alembic migrations.
**Tables:**
1. grants: id (uuid pk), source (text), external_id (text unique), title (text), description (text), agency (text), amount_min (int), amount_max (int), deadline (timestamp), posted_at (timestamp), url (text), eligibility (jsonb), categories (text[]), raw_data (jsonb), embedding (vector(1536)), created_at (timestamp default now)
- Indexes: posted_at DESC, deadline ASC, source, embedding (ivfflat for vector similarity)
2-5. [Similar detailed specs for users, lab_profiles, matches, alerts_sent]
```

Step 2: Generate Code with Claude

$ claude-agent 'Read specs/database_schema.md and implement:'

1. *Create backend/models.py with SQLAlchemy models*
2. *Create backend/database.py with connection setup*
3. *Create alembic migration for initial schema*
4. *Add pgvector extension setup'*

Claude Agent Output:

✓ Created backend/models.py (25 lines)
✓ Created backend/database.py (15 lines)
✓ Created alembic/versions/001_initial.py (50 lines)
✓ Tests passed: 5/5

Time Saved: Manual implementation: 4 hours. Claude Agent: 10 minutes. You review in 20 minutes.

5. **grants:** id (uuid), source (text), external_id (text), title (text), description (text), agency (text), amount_min (int), amount_max (int), deadline (timestamp), posted_at (timestamp), url (text), eligibility (jsonb), categories (text[]), raw_data (jsonb), embedding (vector), created_at (timestamp)
6. **users:** id (uuid), email (text), password_hash (text), name (text), institution (text), phone (text), created_at (timestamp), stripe_customer_id (text)
7. **lab_profiles:** id (uuid), user_id (fk), research_areas (text[]), methods (text[]), career_stage (text), past_grants (jsonb), publications (jsonb), orcid (text), profile_embedding (vector), created_at (timestamp)
8. **matches:** id (uuid), grant_id (fk), user_id (fk), match_score (float), reasoning (text), predicted_success (float), created_at (timestamp), user_action (text), user_feedback (jsonb)
9. **alerts_sent:** id (uuid), match_id (fk), channel (text), sent_at (timestamp), opened_at (timestamp), clicked_at (timestamp)

Key Decision: Use pgvector extension for embeddings (cheaper than Pinecone for MVP). Install: CREATE EXTENSION vector;

Event Bus & Job Queue - Agentic Implementation

Spec for Redis Streams + Celery:

$ claude-agent 'Create event-driven infrastructure:'

1. *backend/events.py: Redis Streams wrapper with XADD/XREAD*
2. *Streams: grants:discovered, grants:validated, matches:computed, alerts:pending*
3. *backend/celery_app.py: Celery config with 3 queues (critical, high, normal)*

4. *Priority routing: >90% match → critical queue*
5. *Include retry policies (3 attempts, exponential backoff)'*

Agent Output: Generates complete infrastructure with error handling, monitoring hooks, and tests. You get production-ready code in <15 minutes.

Key Advantage: Agent handles tedious config (consumer groups, acknowledgments, dead letter queues) while you focus on business logic.

Day 3-4: Data Collection Agents - Agentic Build

Strategy: Let Claude Code Agent implement 3 scrapers in parallel. You write high-level specs, agent generates production code with error handling, logging, and tests.

Agent Task 1: Grants.gov RSS Listener

$ claude-agent 'Create agents/discovery/grants_gov_rss.py:'
- Subscribe to https://www.grants.gov/rss/GG_NewOps.xml
- Poll every 5 min using feedparser
- Extract: opportunity_id, title, agency, close_date, link
- Fetch full details via Grants.gov API: GET https://api.grants.gov/v1/api/search2
- Publish to Redis stream grants:discovered
- Store last_processed_id in Redis to avoid duplicates
- Add structured logging with timestamps
- Handle rate limits (3 req/sec) and network errors
- Register as Celery beat task to run every 5 min'

Agent Work: Generates ~150 lines of production code in 3 minutes. Includes retry logic, exponential backoff, circuit breaker pattern, comprehensive error handling.

Agent Task 2: NSF API + NIH Scraper (Parallel)

Run two agent tasks simultaneously:

Terminal 1:

$ claude-agent 'Create agents/discovery/nsf_api.py with NSF Award Search API integration. Query https://www.research.gov/awardapi-service/v1/awards.json, filter by award_date, paginate 25/page, store last_check_time in Redis. Celery task every 15 min.'

Terminal 2:

$ claude-agent 'Create agents/discovery/nih_scraper.py: Scrape https://grants.nih.gov/funding/searchguide/ with Playwright. Compute SHA-256 hash of HTML, detect changes. On change: extract all FOA links using Claude API with prompt: Extract grant opportunities from this HTML. Return JSON array with foa_number, title, deadline, url. Run every 30 min.'

Parallel Execution: Both agents work simultaneously. Total time: 8 minutes for both scrapers vs 6 hours manual implementation.

Agent-Powered Testing

$ claude-agent 'Create tests/test_discovery.py:'
- Mock RSS feed responses
- Test duplicate detection logic
- Verify Redis stream publishing
- Test error handling (API down, malformed data)'

Result: Agent generates pytest tests with 90% code coverage. You run pytest, verify all pass, move on.

Day 3-4 Summary: 3 production scrapers + tests built in 6 hours total (vs 2-3 days manual). You spend time reviewing code quality, not writing boilerplate.

Day 5-7: Validation & Storage Pipeline

Curation Agent

Consumer: Read from 'grants:discovered' stream

Quality Check:

10. LLM validation prompt: 'Is this a legitimate research grant? Check: (1) Has title and description, (2) Has deadline, (3) Not expired, (4) For research (not community grants). Return JSON: {is_valid: bool, quality_score: 0-100, issues: []}'
11. If quality_score < 70: log to manual_review table, skip for now
12. If valid: continue to enrichment

Data Enrichment:

13. Extract missing fields via LLM if needed
14. Categorize: 'Assign research categories to this grant: {title, description}. Return array of 2-5 categories from: [Biomedical, Computer Science, Physics, Chemistry, Social Sciences, Engineering, etc.]'
15. Generate embedding: Use OpenAI text-embedding-3-small on 'title + description'
16. Store in grants table with embedding

Deduplication:

17. Query database for grants with similar title (Levenshtein distance < 3) OR same external_id from different source
18. If potential duplicate: Ask LLM 'Are these the same grant?' with both entries
19. If duplicate: merge entries (keep most complete version), update confidence score

Output: Publish to 'grants:validated' stream

Code Structure: agents/curation/validator.py - Celery worker consuming stream

Testing & Validation

By end of Day 7, you should have:

- Database with 500+ grants from Grants.gov, NSF, NIH
- Event pipeline working: discover → validate → store
- Latency measured: Average time from grant posted → in database = target <10 minutes
- Quality metrics: 90%+ validation accuracy (manually check 20 random grants)

Week 2: Matching & Alerts

Goal: Match grants to users and send instant alerts via email/SMS within 60 seconds of validation

Day 8-10: Matching Engine

Profile Building

Manual Onboarding Form (for MVP):

20. Create simple Google Form with 8 questions: Name, Email, Institution, Research areas (text), Methods/Techniques (text), Career stage (dropdown), Past grants (optional text), ORCID (optional)
21. Responses go to Google Sheets, export CSV
22. Manual script to import CSV → users + lab_profiles tables

Profile Embedding Generation:

23. Combine profile fields: research_areas + methods + past_grants summary
24. Generate embedding with OpenAI text-embedding-3-small
25. Store in lab_profiles.profile_embedding (pgvector column)

Code Structure: scripts/import_profiles.py - Run manually for MVP

Matching Agent (TF-IDF + Semantic)

Consumer: Read from 'grants:validated' stream

Matching Algorithm (Hybrid):

26. **Vector Similarity:** Use pgvector to find top 50 users by cosine similarity: SELECT user_id, 1 - (profile_embedding <=> grant_embedding) AS similarity FROM lab_profiles WHERE similarity > 0.6 ORDER BY similarity DESC LIMIT 50
27. **LLM Re-ranking:** For top 20 matches, use Claude to compute detailed match: 'Given this lab profile: {profile} and this grant: {grant}, evaluate fit. Return JSON: {match_score: 0-100, reasoning: text, key_strengths: [], concerns: [], predicted_success: 0-100}'
28. **Final Score:** Weighted average: 40% vector similarity + 60% LLM match_score

Storage: Insert into matches table with match_score, reasoning, predicted_success

Output: For each match with score >70, publish to 'matches:computed' stream

Optimization: Batch LLM calls (5 matches per prompt) to reduce latency and cost. Target: Match all users in <60 seconds for typical grant.

Code Structure: agents/matching/matcher.py - Celery worker with high priority

Day 11-14: Alert Delivery System

Delivery Agent

Consumer: Read from 'matches:computed' stream

Smart Routing Logic:

29. **Critical (>95% match + <14 days deadline):** SMS + Email + Slack (if connected)
30. **High (85-95% match):** Email + Slack
31. **Medium (70-85% match):** Email only (batched in digest if >3 medium matches per day)

Email Generation (Personalized):

32. LLM prompt: 'Write a personalized alert email for {user_name} about this grant: {grant}. Match score: {score}. Reasoning: {reasoning}. Tone: Professional but friendly. Include: (1) Why this is a great fit, (2) Key deadline info, (3) Strategic recommendations, (4) CTA to view full details. Keep under 200 words.'
33. Generate subject line: 'Write an engaging subject line (under 50 chars) for this grant alert: {grant_title}'
34. Send via SendGrid API with tracking (opens, clicks)

SMS Generation (Critical only):

35. Template: 'GrantRadar Alert: {grant_title} matches your research {match_score}%. Deadline: {deadline}. View: {short_url}'
36. Send via Twilio API, track delivery status

Tracking & Logging:

37. Insert into alerts_sent table: match_id, channel, sent_at
38. Set up SendGrid webhooks to track opens/clicks → update opened_at, clicked_at
39. Log latency: grant_posted_at → alert_sent_at (target <5 minutes)

Code Structure: agents/delivery/alerter.py - Celery worker with critical priority

Integration Testing

End-to-end test flow:

40. Manually insert test grant into grants table
41. Publish to 'grants:validated' stream
42. Verify matching agent processes it
43. Verify alerts sent to test users
44. Measure total latency (should be <2 minutes)

Week 3: Product & Launch

Goal: Build minimal web dashboard, add payment processing, recruit 10 paying beta users

Day 15-17: Web Dashboard (React)

Frontend Architecture

Tech Stack: React + Tailwind CSS, deployed on Vercel

Key Pages (5 total):

45. **Landing Page:** Hero section with value prop, pricing ($200/mo for beta), testimonials section (leave empty for now), CTA to sign up
46. **Sign Up / Login:** Email/password auth (use Supabase Auth for easy setup), OAuth optional for MVP
47. **Dashboard:** Stats bar (new grants, high matches, deadlines), Grant list (infinite scroll), Filter tabs (All/Federal/Foundation)
48. **Grant Detail Page:** Full grant info, match score with reasoning, action buttons (Save / Dismiss), similar grants section
49. **Settings:** Profile editing, notification preferences, billing

WebSocket Live Updates

Implementation:

50. Backend: Socket.io server (socketio-python) integrated with FastAPI
51. Frontend: Socket.io client, connect on dashboard mount
52. Events: 'new_match' (when grant matched to user), 'deadline_soon' (3 days before deadline)
53. UI: Toast notification appears top-right with animation, counter updates in real-time

API Endpoints (FastAPI)

Build RESTful API:

- GET /api/matches - List user's matches (paginated, filtered)
- GET /api/grants/{id} - Grant details
- POST /api/matches/{id}/action - Save/dismiss grant
- GET /api/profile - Get user profile
- PUT /api/profile - Update profile
- GET /api/stats - Dashboard stats (counts, deadlines)

Authentication: JWT tokens, middleware to verify on protected routes

Day 18-19: Payments & Onboarding

Stripe Integration

54. Set up Stripe account, get API keys
55. Create product: 'GrantRadar Beta' at $200/month
56. Implement checkout flow: POST /api/checkout/session creates Stripe checkout session, redirect to Stripe hosted page
57. Webhook handler: Listen for checkout.session.completed → create user account, send welcome email
58. Customer portal: Link to Stripe customer portal for subscription management

Onboarding Flow

59. **Step 1:** Sign up → Email verification
60. **Step 2:** Payment (Stripe checkout)
61. **Step 3:** Profile setup (8 questions in-app form): Research areas (multi-select + free text), Methods/techniques (free text), Career stage (dropdown), Past grants (optional), ORCID (optional)
62. **Step 4:** Notification preferences (email, SMS, Slack webhook)
63. **Step 5:** Complete → Trigger profile embedding generation → Start matching immediately

Welcome Email: Sent via SendGrid: 'Welcome to GrantRadar! Your profile is being analyzed. You'll receive your first grant matches within 2 hours (or immediately if we find a perfect fit).'

Day 20-21: Testing & Launch

Manual Testing Checklist

- Sign up flow: Create account → Pay → Complete onboarding
- Alert delivery: Manually trigger grant → verify email received within 5 min
- Dashboard: Verify grants load, filters work, WebSocket updates appear
- Match quality: Review 20 random matches, verify >80% are relevant
- Mobile: Test on phone (responsive design)
- Performance: Dashboard loads in <2 seconds

Beta Recruitment Strategy

64. **LinkedIn Outreach:** Target: Lab managers, postdocs, early-career PIs at top 20 universities. Message template: 'Hi {Name}, I built a tool that monitors 60+ grant sources in real-time and alerts you within minutes of perfect-fit opportunities. Beta users get lifetime 50% off ($100/month instead of $200). Interested in trying it?' Goal: Send 50 messages, convert 20% = 10 users.
65. **Reddit Posts:** r/GradSchool, r/AskAcademia - 'I built a real-time grant monitoring tool. Beta testers needed.' Include demo video.
66. **University Listservs:** Email research development offices at 10 universities with product demo
67. **Personal Network:** Reach out to any professors, researchers you know. Ask for intros to lab managers.

Launch Day Checklist

- All scrapers running on schedule (verify logs)
- Database has 1,000+ grants
- Monitoring dashboard set up (use UptimeRobot for uptime, Sentry for errors)
- Stripe webhooks tested in production
- Landing page live with pricing
- DNS configured, SSL certificate active
- Support email set up (support@grantradar.com)

Post-Launch: Week 4+ Priorities

Immediate Improvements (Week 4-6)

68. **Add Top 10 Foundations:** Build scrapers for CZI, Gates, Simons, HHMI, Sloan, Wellcome, Moore, Arnold, Templeton, Kavli
69. **Email Parsing:** Set up inbox, subscribe to foundation newsletters, parse for grant announcements
70. **Prediction Agent V1:** Scrape 5 years of historical data, build basic time-series prediction for top 20 programs
71. **User Feedback Loop:** Add thumbs up/down on each match, collect data to improve matching
72. **Analytics Dashboard:** Build internal dashboard to monitor: alert latency, match accuracy, user engagement, churn signals

Optimization Targets

- Alert Latency: <3 min (from current 5 min)
- Match Accuracy: >90% (from 85%)
- Coverage: 70+ sources (from 30)
- Daily Active Users: >40%
- Email Open Rate: >70%

Scale Preparation

When you reach 50+ users:

- Migrate from pgvector to Pinecone (better performance at scale)
- Add Redis caching layer for API responses
- Set up auto-scaling for Celery workers
- Implement rate limiting on API endpoints
- Add comprehensive logging and alerting

Appendix A: Agentic Workflow Templates

Overview: These are battle-tested prompt templates for Claude Code Agent. Copy-paste and modify for your specific needs.

Template 1: API Endpoint Generation

claude-agent 'Create FastAPI endpoint in backend/api/grants.py:'
Route: GET /api/matches
Auth: Require JWT token in header
Query params: page (int), filter (enum: all/federal/foundation), min_score (float)
Database: Query matches table joined with grants, filter by user_id from token
Response: Pydantic model MatchListResponse with pagination metadata
Error handling: 401 for invalid token, 500 for DB errors
Include OpenAPI docs and unit tests'

Template 2: React Component

claude-agent 'Create React component frontend/src/components/GrantCard.tsx:'
Props: grant (Grant type), matchScore (number), onSave/onDismiss callbacks
UI: Card layout with Tailwind CSS, gradient match score badge, expandable description
Actions: Save button (green), Dismiss button (gray), View Details link
Animations: Fade in on mount, hover scale effect
Responsive: Stack vertically on mobile
Include Storybook stories for all states'

Template 3: Background Worker

claude-agent 'Create Celery worker agents/matching/matcher.py:'
Consume from Redis stream: grants:validated
For each grant:
1. Query pgvector for top 50 similar profiles (cosine similarity >0.6)
2. Batch call Claude API (5 profiles per request) for detailed scoring
3. Insert matches with score >70 into matches table
4. Publish high-scoring matches (>85) to matches:computed stream
Logging: Latency per grant, total matches computed, LLM token usage
Error handling: Retry on LLM timeout, circuit breaker on repeated failures
Target: Process grant + match all users in <60 seconds'

Template 4: Database Migration

claude-agent 'Create Alembic migration to:'
1. Add user_preferences jsonb column to users table
2. Add notification_channels jsonb column (default: {email: true, sms: false, slack: false})
3. Create index on matches(user_id, created_at DESC)
Include rollback logic in downgrade()'

Best Practices for Working with Claude Code Agent

1. Be Specific About Structure

Bad: *'Build a scraper for NIH'*

Good: *'Create agents/discovery/nih_scraper.py: Scrape https://grants.nih.gov/funding/searchguide/ using Playwright, hash HTML for change detection, parse with Claude API, publish to Redis stream grants:discovered, run as Celery task every 30 min'*

2. Specify Error Handling Explicitly

Always include: 'Handle rate limits, network errors, malformed data. Retry 3x with exponential backoff. Log errors with context.'

3. Request Tests Upfront

End every prompt with: 'Include unit tests with pytest. Mock external APIs. Test happy path + error cases. Aim for 85%+ coverage.'

4. Iterate Incrementally

Don't ask for entire system in one prompt. Build component by component. Test each before moving to next.

5. Review Generated Code

Agent generates production-quality code, but always review for: security issues, performance bottlenecks, edge cases, naming conventions.

Time Savings Calculator

Task	Manual Time	With Agent
Database models + migrations	4 hours	30 minutes
3 scrapers + tests	16 hours	2 hours
Celery + Redis setup	6 hours	45 minutes
Matching agent + LLM integration	8 hours	1.5 hours
Alert delivery system	6 hours	1 hour
6 API endpoints + auth	8 hours	1.5 hours
React components	12 hours	3 hours
TOTAL	**60 hours**	**10.5 hours**

Result: You build the MVP in 3 weeks instead of 8 weeks. Time saved = 5 weeks = $20-40k in opportunity cost.

Appendix B: Tools & Resources

Development Tools

- **Code Editor:** VS Code with Python, React extensions
- **Version Control:** GitHub (private repo)
- **API Testing:** Postman or Insomnia
- **Database Client:** DBeaver or pgAdmin for PostgreSQL

Infrastructure & Hosting

- **Frontend:** Vercel (free tier initially)
- **Backend:** Railway ($20/month) or Render
- **Database:** Railway PostgreSQL or Supabase
- **Redis:** Railway Redis or Upstash
- **Monitoring:** Sentry (errors), UptimeRobot (uptime)

Estimated Costs (Month 1)

- Railway/Render: $20-50
- OpenAI API: $50-100 (embeddings + some LLM calls)
- Anthropic API: $100-200 (Claude for matching logic)
- SendGrid: $15 (up to 40k emails/month)
- Twilio SMS: $20-50 (depends on usage)
- Domain: $12/year
- Stripe fees: 2.9% + 30¢ per transaction

Total: ~$250-450/month before revenue

— END OF BUILD PLAN —