Token Maximization Playbook

Highest Token Burn, Lowest Effort — Paste and Walk Away

The goal: extract maximum value from your Max plan by running sessions that burn massive tokens while requiring minimal babysitting. Each prompt below is designed to be pasted and left alone while Claude works through it.

HOW TOKENS ACTUALLY WORK (So You Can Game It)

What burns tokens:

- Every message you send (input tokens)
- Every response Claude generates (output tokens)
- Every tool call (computer use, file read, file write, bash commands) — EACH ONE is a round trip of tokens
- Reading skill files (SKILL.md files are long — each read burns thousands of tokens)
- Code execution output being fed back to Claude (error messages, test results, build logs)
- File contents being read into context (large files = large token burn)
- Iterative debugging loops (error → fix → re-run → new error → fix = token gold)

The multiplier stack:

```
Base prompt                          ~500 tokens
+ Reading 3 skill files              ~15,000 tokens
+ Reading uploaded files             ~5,000-50,000 tokens
+ Writing 10+ files                  ~20,000-50,000 tokens
+ Running code & reading output      ~10,000-30,000 tokens
+ Debugging & iteration              ~20,000-50,000 tokens
+ Final output generation            ~10,000-30,000 tokens
= Total per session                  ~80,000-225,000 tokens
```

The golden formula:

```
Token Burn = (Skill Files Read) × (Files Created) × (Debug Cycles) × (Output Length)
```

Maximize ALL four dimensions simultaneously.

CATEGORY 1: CODEBASE REFACTORS (Highest Effort-to-Token Ratio)

These are the kings of token burn. Claude reads every file, reasons about the architecture, rewrites everything, runs tests, debugs failures, and iterates. A single prompt can sustain 30+ minutes of autonomous work.

1A. Full Codebase Modernization

```
Read every file in this project. Perform a complete modernization:

1. **Type Safety:**
   - Add TypeScript strict mode (if JS) or comprehensive type hints (if Python)
   - Replace every `any` type with proper types
   - Create type definitions for all data structures
   - Add generics where appropriate

2. **Documentation:**
   - Add JSDoc/docstrings to every function, class, and method
   - Parameters, return types, examples, and edge cases
   - Add inline comments explaining non-obvious logic
   - Generate a comprehensive README.md with: project overview, architecture, setup instructions, API reference, contributing guide

3. **Testing:**
   - Write unit tests for every utility function
   - Write integration tests for every API endpoint
   - Write component tests for every React component (if frontend)
   - Aim for >80% code coverage
   - Run all tests and fix any failures

4. **Code Quality:**
   - Remove all dead code
   - Extract duplicate logic into shared utilities
   - Replace magic numbers/strings with named constants
   - Apply consistent error handling patterns
   - Add input validation to every public function
   - Optimize database queries (N+1, missing indexes, unnecessary joins)

5. **Security Audit:**
   - Check for hardcoded secrets
   - Add input sanitization
   - Verify auth on all protected routes
   - Check for SQL injection, XSS, CSRF vulnerabilities

6. **Performance:**
```

```
   - Add caching where appropriate
   - Lazy load components/modules
   - Optimize imports (tree shaking)
   - Add database indexes for slow queries

After ALL changes, run the full test suite. Fix any failures. Show me a summary of:
- Files modified (count)
- Lines added/removed
- Tests added (count)
- Test pass rate
- Issues found and fixed
```

Why this burns tokens: Claude reads every file (huge input), rewrites them all (huge output), writes tests (more output), runs tests (tool calls + output parsing), debugs failures (iteration loops), and generates a summary. One prompt, 30+ minutes of autonomous work.

1B. Framework Migration

```
Migrate this entire project from [current framework] to [new framework].

Specific migration:
- React class components → functional components with hooks
- OR: JavaScript → TypeScript
- OR: Express → Fastify
- OR: Create React App → Next.js App Router
- OR: REST API → GraphQL
- OR: CSS/SCSS → Tailwind CSS
- OR: Any ORM → Prisma/Drizzle

For EVERY file:
1. Read the current implementation
2. Rewrite it in the new framework/pattern
3. Ensure all functionality is preserved
4. Add types if missing
5. Write tests for the migrated code
6. Run tests and fix failures

Create a MIGRATION_LOG.md documenting every change made, why, and any breaking changes.
```

1C. The "Make It Production-Ready" Prompt

This is the ultimate one-prompt token burner for any existing project:

This project is a prototype. Make it production-ready.

Read every file in the project first. Then systematically:

1. **Error Handling:** Wrap every async operation in try/catch. Add error boundaries for React components. Create a centralized error handler. Add user-friendly error messages for every failure mode. Log errors with structured JSON (timestamp, level, context, stack trace).

2. **Input Validation:** Add zod schemas (or equivalent) for every API endpoint input. Validate every form field on the frontend. Add server-side validation for everything the client validates (never trust the client). Sanitize all user input before database operations.

3. **Authentication & Authorization:** Verify every protected route checks auth. Add role-based access control if applicable. Check that API keys aren't exposed client-side. Add rate limiting to all public endpoints. Add CSRF protection.

4. **Performance:** Run the build and analyze bundle size. Split large components with dynamic imports. Add proper caching headers to API responses. Optimize images (WebP, proper sizing, lazy loading). Add database indexes for common queries. Memoize expensive computations.

5. **Testing:** Write tests for every critical path. Test error states, empty states, loading states. Test auth flows (login, logout, expired session). Test payment flows if applicable. Run all tests, fix failures.

6. **Monitoring:** Add a health check endpoint (/api/health). Add structured logging throughout. Create an error tracking setup. Add performance monitoring (response times, error rates).

7. **Documentation:** Update README with production setup instructions. Create .env.example with all required environment variables. Document the API (endpoint, method, auth requirement, request/response shapes). Create a RUNBOOK.md for common operational tasks.

8. **CI/CD:** Create GitHub Actions workflow: lint, type check, test, build. Add pre-commit hooks for formatting. Create a deployment checklist.

Run every test. Fix every failure. Show me the final state.

CATEGORY 2: DOCUMENTATION GENERATION (Massive Output, Zero Debugging)

Pure output-heavy sessions. Claude reads code and writes mountains of text. Low failure rate means fewer interruptions.

2A. Comprehensive Project Documentation Suite

```
Read every file in this project. Generate the following documentation:

1. **README.md** (comprehensive):
   - Project name, description, and purpose
   - Screenshots/demo section (describe what screenshots should show)
   - Tech stack with version numbers
   - Prerequisites and system requirements
   - Installation (step-by-step, copy-pasteable commands)
   - Configuration (every environment variable explained)
   - Usage guide with examples
   - API reference (if applicable)
   - Project structure explained
   - Contributing guidelines
   - Troubleshooting common issues
   - License

2. **ARCHITECTURE.md:**
   - High-level system design (describe with mermaid diagrams)
   - Data flow diagrams (mermaid)
   - Database schema (mermaid ERD)
   - API architecture diagram
   - Component hierarchy (if frontend)
   - Authentication flow diagram
   - Deployment architecture diagram
   - Key design decisions and why they were made

3. **API_REFERENCE.md** (if the project has an API):
   - Every endpoint documented:
     - Method + Path
     - Description
     - Authentication requirements
     - Request headers
     - Request body (with TypeScript interface)
     - Response body (with TypeScript interface)
     - Error responses
     - Example curl commands
     - Example response JSON

4. **DEVELOPMENT.md:**
   - Local development setup
   - How to run tests
```

- How to add a new feature (step-by-step template)
- How to add a new API endpoint
- How to add a new database migration
- Code style guide
- Git workflow (branching, commits, PRs)
- Environment setup for each team member role

5. **DEPLOYMENT.md:**
- Production deployment steps
- Environment variables for production
- Database migration process
- Rollback procedure
- Monitoring and alerts setup
- Scaling considerations
- Disaster recovery plan

6. **CHANGELOG.md:**
- Read git history (if available) and generate a formatted changelog
- Group by feature, fix, refactor

Create ALL files. Each should be thorough — not placeholder content. Total output should be 3000+ lines across all docs.

2B. Generate Docs for Every Script You Own

I'm uploading [X] Python scripts that automate various tasks. For EACH script, generate:

1. A detailed header docstring explaining:
- What the script does
- When/why to use it
- Input requirements
- Output produced
- Dependencies
- Example usage

2. Docstrings for every function

3. A dedicated README section covering:
- Purpose
- Setup instructions
- Configuration options
- Usage examples with sample input/output
- Error handling and troubleshooting
- Scheduling recommendations (if it should run on a schedule)

4. A mermaid flowchart showing the script's logic flow

Then create a MASTER_README.md that:
- Lists all scripts with one-line descriptions
- Groups them by category (data processing, reporting, automation, etc.)
- Shows dependencies between scripts
- Provides a "Quick Start" section for each
- Includes a decision tree: "Which script should I use for X?"

Finally, create a mermaid diagram showing how all the scripts relate to each other and the systems they interact with.

CATEGORY 3: BULK DATA TRANSFORMATION (File I/O Token Multiplier)

Reading large files + processing + generating reports = massive token burn.

3A. Financial Data Analysis Pipeline

Read the uploaded CSV file(s). Perform a comprehensive analysis:

1. **Data Profiling:**
   - Row count, column count
   - Data types per column
   - Missing values per column (count and %)
   - Unique values per column
   - Statistical summary (mean, median, std, min, max, quartiles) for numeric columns
   - Top 10 most common values for categorical columns
   - Detect anomalies and outliers (values >3 standard deviations from mean)

2. **Data Cleaning:**
   - Handle missing values (document strategy: fill, drop, or interpolate for each column)
   - Standardize formats (dates, currency, percentages)
   - Remove duplicates
   - Fix data type issues (strings that should be numbers, etc.)
   - Create a cleaned version of the dataset
   - Document every change in a CLEANING_LOG.md

3. **Analysis & Insights:**
   - Time series analysis if date column exists (trends, seasonality, growth rates)
   - Correlation matrix between numeric columns
   - Group-by analysis for key categorical columns
   - YoY / MoM / QoQ comparisons if applicable

- Top/bottom performers by relevant metrics
- Distribution analysis for key metrics

4. **Visualizations:**
- Create a React/HTML dashboard with charts (use Recharts or Chart.js):
- Line chart: key metrics over time
- Bar chart: category comparisons
- Pie chart: composition breakdowns
- Heatmap: correlation matrix
- Scatter plot: relationship between two key variables
- Box plot: distribution of key metrics
- Each chart should be interactive (hover for details, click to filter)
- Dashboard should be a single HTML file that can be opened in any browser

5. **Report Generation:**
- Create a comprehensive report as both .md and .docx:
- Executive summary (key findings in 3 bullet points)
- Methodology
- Detailed findings with supporting data
- Recommendations based on the data
- Appendix with full statistical tables
- The report should be ready to present to a manager

6. **Export:**
- Cleaned dataset as CSV
- Summary statistics as CSV
- All visualizations as a standalone HTML dashboard
- Report as markdown and Word document

3B. The Multi-Format Report Generator

This one hits 3 skill files simultaneously (docx + xlsx + pptx):

```
Using the data I've uploaded, create ALL THREE of the following:

1. **Word Document Report (.docx):**
   - Professional formatting with table of contents
   - Executive summary
   - Detailed analysis with tables and charts
   - Recommendations section
   - Appendix with raw data tables
   - Header/footer with page numbers
   - Cover page

2. **Excel Dashboard (.xlsx):**
   - Raw data sheet
   - Pivot table analysis sheet
   - Summary statistics sheet
   - Charts sheet with 5+ visualizations
   - Conditional formatting on key metrics
   - Data validation on input fields
   - Named ranges for key data areas
   - Formulas for KPIs

3. **PowerPoint Presentation (.pptx):**
   - Title slide
   - Agenda slide
   - 8-10 content slides covering key findings
   - Charts and data visualizations on each slide
   - Key takeaways slide
   - Next steps / recommendations slide
   - Professional design with consistent theme

All three documents should tell the same story with the same data, just formatted for different audiences and use cases.
```

Why this is a token monster: Claude reads 3 separate SKILL.md files (docx, xlsx, pptx), reads the data file, generates 3 complex output files with different formatting requirements, runs code to create each one, and debugs any issues. Easily 100K+ tokens.

CATEGORY 4: FULL-STACK APP GENERATION (One-Shot Builds)

Single prompts that generate entire applications. The more specific the requirements, the more code Claude writes.

4A. Complete Inventory Management System

Build a complete inventory management system from scratch. Every file, fully functional.

Tech Stack: Next.js 14 App Router, TypeScript, Tailwind CSS, shadcn/ui, Prisma ORM, SQLite (for portability), NextAuth.js

Features — implement ALL of these:

1. **Authentication:**
   - Email/password login and registration
   - Password hashing with bcrypt
   - Session management
   - Protected routes (redirect to login if not authenticated)
   - "Forgot password" flow (mock — just show the UI)

2. **Dashboard:**
   - Total inventory value
   - Items low in stock (below reorder point)
   - Items out of stock
   - Recent activity feed (last 10 actions)
   - Quick stats cards: total items, total categories, items added this week
   - Charts: inventory value by category (pie), stock levels over time (line)

3. **Inventory CRUD:**
   - Add item: name, SKU (auto-generated), category, quantity, unit cost, selling price, reorder point, supplier, location, image URL, notes
   - Edit item: inline editing on the table or modal
   - Delete item: confirmation dialog, soft delete (archived, not destroyed)
   - Bulk actions: select multiple → delete, change category, export

4. **Search & Filtering:**
   - Full-text search across name, SKU, category, supplier
   - Filter by: category, stock status (in stock / low / out), price range, supplier
   - Sort by: name, quantity, value, date added, last updated
   - Pagination: 25 items per page with page controls

5. **Categories:**
   - CRUD for categories
   - Nested categories (parent/child)
   - Color-coded category badges

6. **Stock Management:**

- "Receive Stock" action: add quantity to existing item, log the receipt
- "Ship Stock" action: subtract quantity, log the shipment
- Stock history per item: table showing all additions/subtractions with dates and reasons
- Automatic low-stock alerts (highlight items below reorder point)

7. **Reporting:**
- Inventory valuation report (total cost, total retail value, profit margin)
- Stock movement report (items received/shipped over time period)
- Low stock report (all items below reorder point)
- Export any report as CSV

8. **Settings:**
- Company name and logo
- Default currency
- Low stock threshold (global default)
- Data export (full database as JSON)
- Data import (from JSON or CSV)

9. **Database:**
- Prisma schema with all models and relations
- Seed script with 50+ realistic sample items across 8 categories
- Migration setup

10. **Quality:**
- Input validation with zod on all forms and API routes
- Error handling on every async operation
- Loading skeletons on every data-fetching component
- Empty states for every list
- Toast notifications for all actions
- Responsive design (mobile-friendly)
- Keyboard shortcuts: Ctrl+K for search, N for new item

Build EVERY file. Run the app. Seed the database. Show me the working dashboard with seed data loaded.

4B. Project Management Tool (Kanban)

Build a complete project management tool with Kanban board from scratch.

Tech Stack: Next.js 14 App Router, TypeScript, Tailwind CSS, shadcn/ui, Prisma, SQLite

Features — implement ALL:

1. **Kanban Board:**

- Columns: To Do, In Progress, Review, Done (customizable)
- Drag-and-drop cards between columns (use @hello-pangea/dnd or similar)
- Cards show: title, assignee avatar, priority badge, due date, tag badges
- Click card to open detail modal
- Create new card inline (click "+" in any column)
- Smooth animations on drag

2. **Task Detail Modal:**
- Title (editable inline)
- Description (markdown editor with preview)
- Assignee (dropdown of team members)
- Priority: Urgent / High / Medium / Low (colored badges)
- Due date (date picker, highlight overdue in red)
- Tags (multi-select, custom colors)
- Subtasks (checkbox list, progress bar)
- Comments thread
- Activity log (who changed what, when)
- Attachments (file upload — store file names, mock the actual upload)
- Time tracking (start/stop timer, log manual hours)

3. **Project Management:**
- Multiple projects
- Project dashboard: progress bar, task breakdown by status, overdue count
- Project settings: name, description, members, columns

4. **Views:**
- Kanban board (default)
- List view (sortable table)
- Calendar view (tasks on due dates)
- Timeline/Gantt view (horizontal bars showing task duration) — even a basic version

5. **Filtering & Search:**
- Filter by: assignee, priority, tag, due date range
- Search across all task titles and descriptions
- Save filter presets

6. **Team:**
- Add team members (name, email, avatar, role)
- Assign tasks to members
- Workload view: how many tasks per person, who's overloaded?

7. **Notifications:**
- In-app notification bell
- Notify when: assigned a task, task due tomorrow, mentioned in comment

- Mark as read/unread

8. **Seed Data:**
   - Create 3 projects with 30+ tasks across all statuses
   - 5 team members
   - Realistic content (not "Task 1", "Task 2" — actual project work items)

Build everything. Run it. Show me the Kanban board with seed data loaded and a card being dragged.

CATEGORY 5: AI-POWERED ARTIFACTS (Double Token Burn)

These burn tokens on the outer conversation AND make nested API calls to Claude from within the artifact. Maximum token efficiency.

5A. AI Document Analyzer

Create a React artifact that lets users upload a document (PDF, text, or paste content) and uses the Claude API to analyze it in multiple ways.

Features:
1. **Input:** Text area for pasting content OR file upload (PDF/text)
2. **Analysis Modes** (tabs, user picks one or runs all):
   - **Summary:** Generate a 3-paragraph executive summary
   - **Key Points:** Extract the 10 most important points as bullet points
   - **Action Items:** Extract every actionable item with who/what/when
   - **Sentiment Analysis:** Overall tone, section-by-section sentiment, key phrases driving sentiment
   - **Simplify:** Rewrite the document at a 5th-grade reading level
   - **Critique:** Identify logical gaps, unsupported claims, and areas needing more evidence
   - **Q&A Generator:** Generate 15 quiz questions with answers based on the content
   - **Translation:** Translate to Spanish, French, German, or Japanese (user selects)

3. **Implementation:**
   - Each analysis mode makes a separate Claude API call (model: claude-sonnet-4-20250514)
   - "Run All" button processes all modes sequentially
   - Show progress: "Running Summary... ✓ Running Key Points... ⏳"
   - Cache results so switching tabs doesn't re-run the analysis
   - Display results in clean, readable cards per mode
   - "Export All" button that compiles all analyses into one downloadable text

4. **Chat Follow-Up:**
   - After analysis, show a chat input: "Ask questions about this document"
   - Each chat message sends the original document + conversation history to Claude

- Streaming responses if possible, otherwise show typing indicator

This artifact should use the Claude API for every analysis, creating 8+ API calls for a single document. Include error handling for API failures and rate limits.

5B. AI Data Explorer

Create a React artifact that lets users paste CSV data and explore it with AI assistance.

Features:
1. **Data Input:** Paste CSV text or upload a CSV file
2. **Auto-Analysis:** On upload, automatically:
   - Parse the CSV and show a preview table (first 20 rows)
   - Call Claude API to analyze the data structure and suggest interesting analyses
   - Display column types, missing values, basic stats

3. **Natural Language Queries:**
   - Chat input: "Ask anything about your data"
   - User types questions like:
     - "What's the average revenue by region?"
     - "Show me a trend of monthly sales"
     - "Which customers have the highest lifetime value?"
     - "Are there any outliers in the pricing data?"
   - Claude generates analysis code, runs it mentally, and returns the answer + a chart specification
   - Render charts using Recharts based on Claude's response

4. **Auto-Generated Dashboard:**
   - "Generate Dashboard" button
   - Claude API call: "Given this dataset, generate a JSON specification for the 6 most insightful charts to display"
   - Render all 6 charts in a grid layout
   - Each chart is interactive (hover, click)

5. **Export:**
   - Export analysis as markdown report
   - Export charts as images (if possible)
   - Export filtered/transformed data as CSV

Each interaction with the data makes a Claude API call, so a typical session generates 10-15 nested API calls.

5C. AI Meeting Notes Processor

Create a React artifact for processing meeting notes or transcripts.

1. **Input:** Paste meeting notes, transcript, or recording summary
2. **Processing Pipeline (runs automatically, each step is a Claude API call):**
   - Step 1: Clean and format the raw notes
   - Step 2: Generate structured meeting summary (attendees, topics discussed, decisions made)
   - Step 3: Extract all action items with assignees and deadlines
   - Step 4: Generate follow-up email draft to send to attendees
   - Step 5: Create a list of unanswered questions that need follow-up
   - Step 6: Suggest agenda items for the next meeting based on open items
   - Step 7: Rate the meeting productivity (1-10) with justification

3. **Display:** Tab for each output, all generated in sequence
4. **Edit & Regenerate:** User can edit any output and ask Claude to regenerate based on edits
5. **Export:** Copy any section, or export all as a formatted document

That's 7 Claude API calls per meeting, plus any regeneration requests. Show a progress bar as each step completes.

CATEGORY 6: THE MEGA-COMBOS (Maximum Token Burn)

These combine multiple categories for compounding token usage.

6A. The Triple-File Generator (docx + xlsx + pptx)

I need three deliverables from the attached data. Create ALL THREE:

1. A comprehensive Word document report (.docx) with:
   - Cover page, table of contents, headers/footers
   - Executive summary
   - 5+ sections of detailed analysis with tables
   - Charts embedded in the document
   - Recommendations section
   - Appendix with methodology

2. An Excel workbook (.xlsx) with:
   - Raw data sheet
   - 3 pivot analysis sheets
   - Dashboard sheet with 6+ charts
   - Summary KPI sheet
   - Conditional formatting throughout

3. A PowerPoint presentation (.pptx) with:
   - 12+ slides
   - Data visualizations on each content slide

- Speaker notes for every slide
- Professional design theme
- Agenda and takeaway slides

All three should tell the same story. Start by reading all three skill files (docx, xlsx, pptx), then build each one.

6B. Full Project: Build + Document + Test + Deploy Config

Build a complete [app type] from scratch AND generate full documentation AND write comprehensive tests AND create deployment configuration.

Phase 1 — Build:
[Insert one of the full-stack app prompts from Category 4]

Phase 2 — Document:
After the app is built, generate:
- README.md (comprehensive)
- ARCHITECTURE.md with mermaid diagrams
- API_REFERENCE.md
- CONTRIBUTING.md
- DEPLOYMENT.md

Phase 3 — Test:
Write and run:
- Unit tests for all utility functions
- Integration tests for all API routes
- Component tests for key UI components
- E2E test scenarios documented (if not runnable)
Fix all failures.

Phase 4 — Deploy Config:
- Dockerfile
- docker-compose.yml
- GitHub Actions CI/CD pipeline
- Vercel/Railway deployment config
- Environment variable documentation

This is a 4-phase session. Complete all phases sequentially.

6C. The "Improve Everything" Audit

Feed Claude any existing project and this prompt:

Read every file in this project. Then perform ALL of the following in sequence:

PHASE 1: AUDIT (read and analyze)
- List every file with a 1-line description of what it does
- Identify the tech stack and architecture pattern
- List all dependencies and their versions
- Find all TODO/FIXME/HACK comments
- Identify code smells and anti-patterns
- Find security vulnerabilities
- Find performance bottlenecks
- Rate code quality 1-10 with justification

PHASE 2: FIX (modify files)
- Fix every security vulnerability found
- Fix every performance bottleneck found
- Resolve every TODO/FIXME/HACK
- Add error handling where missing
- Add input validation where missing
- Add TypeScript types where missing
- Remove dead code
- Extract duplicate code into utilities

PHASE 3: TEST (create and run tests)
- Write tests for every function that doesn't have one
- Run all tests
- Fix any test failures
- Report coverage

PHASE 4: DOCUMENT (generate docs)
- Add docstrings/JSDoc to every undocumented function
- Generate README.md
- Generate ARCHITECTURE.md with mermaid diagrams
- Create .env.example

PHASE 5: REPORT
Create a AUDIT_REPORT.md with:
- Summary of findings
- Changes made (count: files modified, lines added/removed, tests added)
- Before/after comparison of code quality score
- Remaining recommendations

Show me the final report.

OPTIMAL SESSION SCHEDULING

Here's how to maximize your daily token budget:

Morning Session (Paste and Walk Away — 45-60 min)

Pick ONE of these high-burn prompts:

- Category 1: Codebase refactor (your existing projects)
- Category 4: Full-stack app generation
- Category 6: Mega-combo

Afternoon Session (Paste and Walk Away — 30-45 min)

Pick ONE of these:

- Category 2: Documentation generation
- Category 3: Data transformation + report
- Category 5: AI-powered artifact

Evening Session (Quick Burns — 15-20 min each)

Stack 2-3 of these quick token burners:

- "Add comprehensive tests to [project]"
- "Generate mermaid diagrams for [project]'s architecture"
- "Create a .pptx presentation explaining [project]"
- "Refactor [specific file] with full type safety, error handling, and tests"

The Nuclear Option (Maximum Possible Burn)

Run this on each of your existing projects:

```
Read every file in this project. Then:
1. Read the docx skill file and generate a comprehensive Word document report about this project
2. Read the xlsx skill file and create an Excel workbook with project metrics, dependency analysis, and code quality scores
3. Read the pptx skill file and create a PowerPoint presentation about this project's architecture
4. Write comprehensive tests for every untested function
5. Add docstrings to every function
6. Create ARCHITECTURE.md with 10+ mermaid diagrams
7. Create a Dockerfile and docker-compose.yml
8. Create a GitHub Actions CI/CD pipeline
9. Generate a detailed CHANGELOG by reading git history
10. Create a SECURITY_AUDIT.md documenting all potential vulnerabilities

Complete ALL 10 tasks in this session.
```

This hits 3 skill files, reads every project file, generates 10+ output files across multiple formats, runs code, and debugs issues. Easily the highest token burn per prompt.

PROJECTS TO FEED IT (From Your Portfolio)

You've built 8 apps across 14 sessions. Each one is refactor/doc/test fodder:

Project	Best Token Burn Prompt	Why
Stock Trading App	Full Modernization (1A)	Largest codebase, most to refactor
Chess Roguelike	Full-Stack Rewrite (1B)	Game logic is complex, lots of tests to write
CRM	Production-Ready (1C)	CRUD apps have tons of endpoints to document/test
Music Visualizer	Documentation (2A)	Creative code needs heavy documentation
PoE Assistant	Audit + Fix (6C)	API integrations = lots of error handling to add
AI Finance Brief	Triple-File (6A)	Generate docs about the product itself
Chess Coach	Test Suite (from 1A)	Game logic has hundreds of edge cases to test
PC Analyzer	Full Docs (2B)	Python + web = document both sides

---

QUICK REFERENCE: TOKEN BURN BY PROMPT TYPE

Prompt Type	Est. Tokens	Time	Effort
Full codebase refactor	150-250K	45-60 min	Paste & walk away
Triple-file generator (docx+xlsx+pptx)	120-200K	30-45 min	Paste & walk away
Full-stack app generation	100-180K	30-45 min	Paste & walk away
AI artifact with nested calls	80-150K	20-30 min	Paste & walk away
Documentation generation	80-120K	20-30 min	Paste & walk away
Data analysis + visualization	60-120K	20-30 min	Paste & walk away
Test suite generation	60-100K	20-30 min	Paste & walk away
Single file refactor	20-40K	10-15 min	Paste & walk away

Daily target: 2-3 sessions = 300-600K tokens/day Monthly at this pace: 9-18M tokens

The key insight: the prompts that burn the most tokens are also the ones that produce the most useful output. You're not wasting tokens — you're getting production-ready code, comprehensive docs, and thorough tests. The token burn IS the value.