Week 1: Foundation

Day 1

Goal: Set up your development environment, understand Git workflows, and learn to direct Claude Code while verifying its outputs.

Shorts to watch

- **Fireship - VS Code in 100 Seconds**
- **Git Explained in 100 Seconds**
- **Cursor 3 minute demo - the most popular AI code editor**
-

Environment Setup

- Install iterm 2, homebrew on mac (we will install claude code via homebrew not directly, else we wont be able to upgrade automatically)
- Install Cursor IDE, set up Claude Code CLI within Cursor terminal
- Install GitHub Desktop app (for visual git operations without command line)
- Create GitHub account
- Learn terminal basics: `cd`, `ls`, `mkdir`, `touch`, `rm`, tab completion
- Learn about brew package manage, install it (use claude desktop app for help)
- **Task:** Navigate to a folder, create a file, and delete it using terminal, change folders and navigate using tab completion

Git Fundamentals

- Complete GitHub Skills: Introduction to GitHub (click "Start Course" in readme)
- Complete GitHub Skills: Review Pull Requests
- Learn branching, merging, resolving conflicts (ask Claude in browser to explain with diagrams)
    - **Learn Git Branching (interactive visualization)**
- Practice using GitHub Desktop for: viewing changes, staging commits, switching branches, pushing/pulling
- **Task**: Can you explain to someone what a branch is? What a pull request does? When you'd use merge vs rebase?

Python Awareness (Not Coding)

You won't write Python, Claude Code will. But you need to recognize what you're looking at to verify outputs.

Ask Claude (browser or desktop) to explain these concepts at a high level:

- How to run a Python file (`python filename.py` or `python3 filename.py`)
- What `import` statements do (bringing in external tools)
- What `if __name__ == "__main__":` means (entry point of the program)
- Indentation matters in Python (vs brackets `{}` in other languages)
- What a function looks like (`def function_name():`)
- What `pip install` does (installing packages)
- What `requirements.txt` is (list of packages needed)

Task: When Claude Code generates a Python file, can you identify: the imports, the main function, where the program starts running?

Mini-Projects

Project 1: CLI Tool with File Operations

Ask Claude Code to build a command-line tool that:

- Takes a folder path as input
- Scans all files in that folder
- Outputs a summary: total files, total size, largest file, file types breakdown
- Saves the report to a text file

Your job:

1. Give Claude Code the requirements above
2. Run the generated script on a test folder
3. Verify the output matches reality (manually count files in a small folder)
4. Push to GitHub using GitHub Desktop
5. Check that your repo shows the files on github.com

Project 2: Git Workflow Practice

Do this sequence 3 times until muscle memory:

1. Create a new branch from github desktop app

2. Ask Claude Code to add a new feature to your Project 1 (e.g., "add option to filter by file extension")
3. Commit changes with a descriptive message
    4. Push branch to GitHub
    5. Open a Pull Request on github.com
    6. Review the changes (look at the dif)
    7. Merge the PR
    8. Pull latest changes to your local machine

Your job:

- Verify each step worked by checking GitHub Desktop and github.com
- On the 3rd attempt, intentionally create a merge conflict (edit same line in two branches) and resolve it

Day 2

Project 3: Multi-File Python Project

Ask Claude Code to build a "Plivo Account Health Checker" that:

- Reads Plivo API credentials from a `.env` file (not hardcoded)
- Calls Plivo API to get account balance
- Calls Plivo API to get recent message logs
- Outputs a formatted report with: current balance, messages sent today, any failed messages
- Includes proper error handling (what if API is down? what if credentials wrong?)

Your job:

1. Verify the project structure makes sense (separate files for config, API calls, main logic)
2. Verify `.env` file is in `.gitignore` (credentials should never go to GitHub, ask claude LLM on what is gitignore and why it matters)
3. Run the script with your real Plivo credentials
4. Intentionally break it (wrong API key) and verify error handling works
5. Push to GitHub, confirm `.env` is NOT visible in the repo

End of Project Checklist

- Cursor IDE installed and Claude Code working in terminal

- GitHub Desktop installed and connected to account
- Can navigate terminal comfortably
- Completed both GitHub Skills courses
- Can explain Git concepts (branch, PR, merge) verbally
- Can identify parts of a Python file (imports, main, functions)
- Project 1 running and verified☐Completed 3 full branch → PR → merge cycles
- Project 3 running with real Plivo credentials
- All projects pushed to GitHub (no secrets exposed)

Day 3

Web Servers, Databases & Plivo Voice

Goal: Understand how web servers work, why they're needed for Plivo webhooks, and build a voice IVR with persistent storage.

Shorts to watch first

- **Flask in 100 Seconds - Fireship**
- **Webhooks Explained - Fireship**
- **ngrok in 100 Seconds**
- **Redis in 100 Seconds - Fireship**
- **PostgreSQL in 100 Seconds - Fireship**

Part 1: Flask Fundamentals (Learn from Claude, Verify Understanding)

Ask Claude (browser) to explain these concepts — don't memorize, just understand the "why":

Why Flask vs Plain Python Scripts?

- A Python script runs once and exits
- Flask is a web server — it runs continuously, waiting for requests
- Plivo needs to "call back" your code when something happens (incoming call, SMS received)
- Your code needs to be reachable via a URL — Flask provides that

Core Flask Concepts:

- **Routes:** URLs that your server responds to (e.g., `/answer-call`, `/handle-input`)
- **GET vs POST:** GET retrieves data, POST sends data. Plivo webhooks use POST (sending call details to you)

- **Starting Flask:** `flask run` or `python app.py` — server starts and listens on a port (usually 5000)
- **Request/Response:** Plivo sends a request with call info → your Flask route processes it → returns XML response

Verification: Can you explain to someone why a plain Python script won't work for handling Plivo callbacks?

Part 2: ngrok — Making Your Local Server Reachable

Ask Claude (browser) to explain:

The Problem:

- Your Flask server runs on `localhost:5000` — only your computer can access it
- Plivo's servers are on the internet — they can't reach your `localhost`
- You need a public URL that forwards to your local machine

ngrok's Role:

- ngrok creates a tunnel: `https://abc123.ngrok.io` → `localhost:5000`
- Plivo can now send webhooks to your public ngrok URL
- Your local Flask server receives the request and responds

Why Not Just Deploy?

- During development, you want fast iteration
- Change code → save → test immediately (no deploy wait)
- ngrok is for development only; production uses real servers

Verification: Can you draw (on paper) the flow: Phone call → Plivo → ngrok → your Flask server → response back?

Part 3: Redis & PostgreSQL Setup

Install via Homebrew:

bash

```shell
brew install redis
brew install postgresql@15
brew services start redis
brew services start postgresql@15
```

Install a Database Viewer:

- Install TablePlus (free tier available) or DBeaver (free)
- These let you visually browse your database tables, run queries, see data

Ask Claude (browser) to explain:

Redis — What & Why:

- In-memory data store (very fast, data lives in RAM)
- Use cases: caching, session storage, real-time counters
- Data can expire automatically (TTL - time to live)
- For our IVR: store active call state, track caller through menu navigation

PostgreSQL — What & Why:

- Relational database (data persists to disk)
- Use cases: permanent storage, structured data, queries
- For our IVR: store call logs, caller history, menu configurations

SQLAlchemy — What & Why:

- Python library to interact with databases
- Write Python code instead of raw SQL
- Handles connections, queries, and data mapping

Verification:

- Open TablePlus, connect to your local PostgreSQL (host: localhost, user: your Mac username, no password)
- Can you see the connection works?
- Run `redis-cli ping` in terminal — should return `PONG`

Part 4: Plivo Voice Fundamentals

Set Up:

- Create Plivo account, get Auth ID and Auth Token
- Buy a phone number with voice capability
- Read Plivo Voice Quickstart docs

Ask Claude (browser) to explain:

How Plivo Voice Works:

1. Someone calls your Plivo number
2. Plivo sends a webhook (POST request) to your server with call details
3. Your server responds with XML instructions (what to say, what to do)
4. Plivo executes those instructions

Key XML Elements:

- `<Speak>` — text-to-speech, say something to caller
- `<Play>` — play an audio file
- `<GetDigits>` — wait for caller to press buttons, then send those digits to another webhook
- `<Redirect>` — send call to a different URL for next instructions

Verification: Look at a Plivo XML example in the docs. Can you identify what each element does?

Day 4

Mini-Projects

Project 1: Basic Flask Server with ngrok

Ask Claude Code to build a Flask server that:

- Has a route `/health` that returns `{"status": "ok"}` (GET request)
- Has a route `/webhook-test` that accepts POST, logs whatever data is received, returns `{"received": true}`
- Runs on port 5000

Your job:

1. Run the Flask server
2. Open browser, go to `http://localhost:5000/health` — verify you see the JSON response
3. Start ngrok: `ngrok http 5000`

4. Copy the ngrok URL (e.g., `https://abc123.ngrok.io`)
5. Use Claude Code to write a quick script that sends a POST request to your ngrok URL `/webhook-test`
6. Check your Flask terminal — verify the data was logged

Verification Checklist:

- Flask server running
- Health endpoint works in browser
- ngrok tunnel active
- POST to ngrok URL received by local Flask

Project 2: PostgreSQL + SQLAlchemy Integration

Ask Claude Code to extend your Flask app:

- Add SQLAlchemy with PostgreSQL connection
- Create a `CallLog` model with fields: `id`, `caller_number`, `called_number`, `call_status`, `created_at`
- Add a route `/log-call` (POST) that creates a new call log entry
- Add a route `/call-logs` (GET) that returns all call logs as JSON

Your job:

1. Run the app (it should create the table automatically)
2. Open TablePlus, connect to PostgreSQL, verify `call_logs` table exists
3. Use Claude Code to write a test script that POSTs fake call data to `/log-call`
4. Refresh TablePlus — verify the row appears
5. Hit `/call-logs` in browser — verify you see the data as JSON

Verification Checklist:

- PostgreSQL connection working
- Table visible in TablePlus
- Can insert data via API
- Can read data via API and in TablePlus

Project 3: Redis for Call State

Ask Claude Code to add Redis to your Flask app:

- Connect to local Redis
- Add a route `/start-session/<caller_id>` that stores `{caller_id: {"step": "greeting", "started_at": timestamp}}` in Redis with 30-minute expiry
- Add a route `/get-session/<caller_id>` that retrieves the session
- Add a route `/update-session/<caller_id>/<step>` that updates the step value

Your job:

1. Test the session flow: start → get → update → get
2. Verify in terminal using `redis-cli`:
    - `redis-cli KEYS *` — see your session keys
    - `redis-cli GET <key>` — see the data
    - `redis-cli TTL <key>` — see time remaining before expiry
3. Wait 30 minutes (or set shorter TTL for testing) — verify session auto-deletes

Verification Checklist:

- Can create session via API
- Can see session in redis-cli
- Session has correct TTL
- Session auto-expires

Project 4: Plivo Voice IVR with Full Stack

Ask Claude Code to build a complete IVR system using Flask + Plivo Python SDK + Redis + PostgreSQL:

Requirements:

- Route `/answer` — Plivo calls this when someone dials in
    - Store call session in Redis (caller number, current step = "main_menu")
    - Log call start in PostgreSQL
    - Return XML: "Welcome to Acme Corp. Press 1 for Sales, Press 2 for Support, Press 3 to hear your caller ID"
- Route `/handle-input` — Plivo calls this with digit pressed
    - Read current session from Redis
    - Update session step based on input
    - If 1: Return XML "Connecting to sales. Goodbye." + update call log with status "routed_sales"
    - If 2: Return XML "Connecting to support. Goodbye." + update call log with status "routed_support"

    - If 3: Read caller's phone number from Plivo request, return XML speaking their number back
    - Invalid input: Return XML "Invalid option" and redirect back to `/answer`
- Route `/call-history/<phone_number>` (GET) — Returns all past calls from that number as JSON

Your job:

1. **Setup:**
    - Get your ngrok URL
    - In Plivo console, set your phone number's Answer URL to `https://your-ngrok.io/answer`
2. **Test the flow:**
    - Call your Plivo number from your phone
    - Verify you hear the greeting
    - Press 1 — verify you hear "Connecting to sales"
    - Call again, press 3 — verify it reads your phone number back
    - Call again, press 9 — verify it says "Invalid option" and replays menu
3. **Verify data layer:**
    - Check Redis (`redis-cli`) — see active call session during call
    - Check PostgreSQL (TablePlus) — see call log entries after each call
    - Hit `/call-history/<your-phone>` — verify your calls appear
4. **Test edge cases:**
    - What happens if caller hangs up mid-call? (Check if call log captures this)
    - What happens if Redis is down? (Stop Redis, make a call — does it fail gracefully?)

Verification Checklist:

- Answer URL configured in Plivo console
- Can call and hear greeting
- All 3 menu options work correctly
- Invalid input handled properly
- Sessions visible in Redis during active calls
- Call logs written to PostgreSQL
- Call history API returns correct data

End of Project Checklist

- Can explain why Flask is needed for webhooks (vs plain scripts)
- Can explain ngrok's role in development
- Redis installed and running (redis-cli ping returns PONG)
- PostgreSQL installed and running

- TablePlus connected to PostgreSQL
- Project 1: Flask + ngrok working
- Project 2: PostgreSQL read/write verified in TablePlus
- Project 3: Redis sessions working with TTL
- Project 4: Full IVR working end-to-end
- Project 4: Can see data in both Redis and PostgreSQL
- Made at least 5 test calls to verify all menu paths

Day 5

Cloud Deployment with Vercel

Goal: Deploy your IVR app to production using Vercel with managed Redis and Postgres, all provisioned directly through Vercel's dashboard.

Shorts to Watch First

- Vercel in 100 Seconds - Fireship
- Serverless Functions in 100 Seconds - Fireship

Prerequisites: Sign Up for Vercel

Service	Sign Up URL	What You Need
Vercel	vercel.com/signup	Connect your GitHub account

That's it — Redis and Postgres are provisioned directly through Vercel's Storage tab. No separate signups needed.

Your job:

- Create Vercel account
- Connect your GitHub account during signup

Verification: Can you log into Vercel dashboard and see your GitHub repos?

Mini-Projects

Project 1: Deploy Basic Flask to Vercel

Ask Claude Code to convert your Day 4 Flask app for Vercel:

Requirements:

- Restructure project for Vercel serverless functions
- Create `vercel.json` configuration
- Create an `/api/health` endpoint that returns `{"status": "ok"}`
- Create an `/api/webhook-test` endpoint that accepts POST and returns the received data

Your job:

1. **Restructure the project:**
    - Ask Claude Code to create Vercel-compatible structure
    - Verify it creates `/api` folder with Python files
2. **Install Vercel CLI:**

bash

```
Shell
    npm install -g vercel
```

3. **Deploy:**

bash

```
Shell
    vercel
```

- Follow prompts to link to your Vercel account
- Say "Yes" to link to existing project or create new
- Note the URL: `https://your-project.vercel.app`

4. **Test:**
    - Visit `https://your-project.vercel.app/api/health` in browser
    - Use Claude Code to write a script that POSTs to `/api/webhook-test`
    - Check Vercel dashboard → **Logs** to see the request

Verification Checklist:

- Project restructured for Vercel
- `vercel` CLI installed
- Deployed successfully
- Health endpoint works
- Webhook test endpoint receives POST data
- Can see logs in Vercel dashboard

Project 2: Add Redis via Vercel Storage

Provision Redis through Vercel (no separate signup):

1. Go to vercel.com/dashboard
2. Click on your deployed project
3. Go to **Storage** tab
4. Click **Create Database**
5. Select **Redis** (powered by Upstash)
6. Name it `ivr-sessions`
7. Select region closest to you
8. Click **Create**
9. When prompted, click **Connect to Project** → select your project
10. Vercel automatically adds these environment variables:
    - `KV_REST_API_URL`
    - `KV_REST_API_TOKEN`
    - `KV_URL`

Ask Claude Code to add Redis session management:

Requirements:

- Connect to Redis using `KV_REST_API_URL` and `KV_REST_API_TOKEN` environment variables
- Use the `@upstash/redis` package or HTTP REST API

- Create `/api/start-session` that stores `{"step": "greeting", "started_at": timestamp}` with 30-min TTL
- Create `/api/get-session` that retrieves session by caller ID
- Create `/api/update-session` that updates session step

Your job:

1. **Redeploy to pick up new environment variables:**

bash

```
Shell
vercel --prod
```

2. **Test session endpoints:**
   - POST to `/api/start-session?caller_id=+1234567890`
   - GET `/api/get-session?caller_id=+1234567890` — verify session exists
   - POST to `/api/update-session?caller_id=+1234567890&step=menu_selection`
   - GET again — verify step updated
3. **Verify in Vercel dashboard:**
   - Go to **Storage** → click on your Redis database
   - Click **Data Browser** tab
   - See your session keys and values

Verification Checklist:

- Redis created via Vercel Storage tab
- Connected to your project
- Environment variables auto-configured (check Settings → Environment Variables)
- Session endpoints working
- Can see data in Data Browser
- Sessions expire after TTL

Project 3: Add Postgres via Vercel Storage

Provision Postgres through Vercel (no separate signup):

1. Go to Vercel dashboard → your project → **Storage** tab
2. Click **Create Database**

3. Select **Postgres** (powered by Neon)
4. Name it `ivr-call-logs`
5. Select region (same as your Redis for lowest latency)
6. Click **Create**
7. Click **Connect to Project** → select your project
8. Vercel automatically adds these environment variables:
    - `POSTGRES_URL`
    - `POSTGRES_PRISMA_URL`
    - `POSTGRES_URL_NON_POOLING`
    - `POSTGRES_USER`
    - `POSTGRES_HOST`
    - `POSTGRES_PASSWORD`
    - `POSTGRES_DATABASE`

Ask Claude Code to add Postgres for call logs:

Requirements:

- Connect to Postgres using `POSTGRES_URL` environment variable
- Create `/api/setup-db` that creates the `call_logs` table (run once)
- Create `/api/log-call` (POST) that inserts a call record
- Create `/api/call-logs` (GET) that returns all logs as JSON
- Create `/api/call-history/[phone]` that returns logs for a specific number

Your job:

1. **Redeploy:**

bash

```
Shell
    vercel --prod
```

2. **Initialize database:**
    - Hit `/api/setup-db` once in browser to create the table
    - Should return `{"message": "Table created successfully"}`
3. **Test CRUD operations:**
    - Use Claude Code to write a script that POSTs fake call data to `/api/log-call`
    - GET `/api/call-logs` — verify data appears
    - GET `/api/call-history/+1234567890` — verify filtering works
4. **Verify in Vercel dashboard:**
    - Go to **Storage** → click on your Postgres database

    - Click **Data** tab to browse tables
    - Or click **Query** tab and run: `SELECT * FROM call_logs;`

Verification Checklist:

- Postgres created via Vercel Storage tab
- Connected to your project
- Environment variables auto-configured
- Table created successfully
- Can insert call logs via API
- Can read call logs via API
- Can filter by phone number
- Data visible in Vercel Storage Data tab

Project 4: Full IVR Deployment to Vercel

Ask Claude Code to deploy the complete IVR system:

Requirements:

- All IVR routes as Vercel serverless functions:
    - `/api/answer` — Plivo calls this on incoming call
    - `/api/handle-input` — Plivo calls this with digit pressed
    - `/api/call-history/[phone]` — Returns call history for a number
- Redis for session state (using Vercel's Redis)
- Postgres for call logs (using Vercel's Postgres)
- Plivo SDK for generating XML responses
- Read all credentials from environment variables

Your job:

1. **Add Plivo credentials to Vercel:**
    - Go to Vercel dashboard → your project → **Settings** → **Environment Variables**
    - Add `PLIVO_AUTH_ID` → paste your Auth ID
    - Add `PLIVO_AUTH_TOKEN` → paste your Auth Token
    - Select all environments (Production, Preview, Development)
    - Click **Save**
2. **Deploy:**

bash

```shell
vercel --prod
```

3. **Configure Plivo to use your Vercel URL:**
   - Go to Plivo console → **Phone Numbers** → click your number
   - Set **Answer URL** to: `https://your-project.vercel.app/api/answer`
   - Set **Method** to: `POST`
   - Click **Save**
4. **Test the full flow:**
   - Call your Plivo number from your phone
   - Verify you hear: "Welcome to Acme Corp. Press 1 for Sales..."
   - Press 1 → hear "Connecting to sales. Goodbye."
   - Call again, press 2 → hear "Connecting to support. Goodbye."
   - Call again, press 3 → hear your phone number read back
   - Call again, press 9 → hear "Invalid option" and menu replays
5. **Verify data persistence:**
   - During an active call: Go to Vercel Storage → Redis → Data Browser → see session
   - After calls: Go to Vercel Storage → Postgres → Data tab → see call log entries
   - Hit `/api/call-history/<your-phone>` in browser → see your calls as JSON
6. **Test with an external caller:**
   - Have a colleague or friend call your Plivo number
   - Verify it works for them too
   - Check that their call appears in the call logs

Verification Checklist:

- Plivo credentials added to Vercel Environment Variables
- IVR deployed to production URL
- Plivo Answer URL updated to Vercel (no more ngrok!)
- Can call and hear greeting
- Press 1 → Sales message works
- Press 2 → Support message works
- Press 3 → Phone number readback works
- Press 9 → Invalid input handling works
- Sessions visible in Redis Data Browser during calls
- Call logs visible in Postgres Data tab
- `/api/call-history` endpoint returns correct data
- Works for external callers

Project 5: Production Health Check & Monitoring

Ask Claude Code to add production-ready health monitoring:

Requirements:

- `/api/health` endpoint that checks:
  - Redis connectivity (try to ping)
  - Postgres connectivity (try a simple query)
  - Returns: `{"status": "healthy", "redis": "ok", "postgres": "ok", "timestamp": "..."}`
  - If any service fails: `{"status": "unhealthy", "redis": "error", "postgres": "ok", "error": "..."}`
- Proper error handling in all IVR routes that logs errors (visible in Vercel Logs)

Your job:

1. **Deploy and test health check:**

bash

```
Shell
vercel --prod
```

- Hit `/api/health` — verify all services show "ok"

2. **Explore Vercel dashboard monitoring:**
   - **Deployments** tab: See all your deploys, rollback if needed
   - **Logs** tab: See real-time request logs and errors
   - **Analytics** tab: See traffic patterns (may need to enable)
   - **Storage** tab: Monitor database usage
3. **Test error scenarios:**
   - Make a call and check Vercel Logs — see the request/response flow
   - Intentionally cause an error (e.g., invalid input) — verify it's logged

Verification Checklist:

- Health endpoint checks Redis and Postgres connectivity
- Health endpoint returns proper status
- Can view real-time logs in Vercel dashboard
- Can see deployment history
- Understand how to rollback if needed
- Errors are logged and visible